O'REILLY®

TURING 图灵程序设计丛书

全彩印刷



R数据科学

R for Data Science

摒弃其他R语言工具书从头到尾讲统计的陋习

从实用的R包出发，带你重新认识R和数据科学

[新西兰] 哈德利·威克姆 [美] 加勒特·格罗勒芒德 著

陈光欣 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

译者介绍

陈光欣

毕业于清华大学并留校工作，主要兴趣为数据分析与数据挖掘。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

图灵程序设计丛书

R数据科学

R for Data Science

[新西兰] 哈德利·威克姆 [美] 加勒特·格罗勒芒德 著
陈光欣 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（C I P）数据

R数据科学 / (新西兰) 哈德利•威克姆
(Hadley Wickham) , (美) 加勒特•格罗勒芒德
(Garrett Grolemund) 著 ; 陈光欣译. -- 北京 : 人民
邮电出版社, 2018.8
（图灵程序设计丛书）
ISBN 978-7-115-48639-4

Ⅰ. ①R… Ⅱ. ①哈… ②加… ③陈… Ⅲ. ①程序语
言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2018)第124271号

内 容 提 要

本书的目标是教会读者使用最重要的数据科学工具，从而为实施数据科学奠定坚实的基础。读完本书后，你将掌握 R 语言的精华，并能够熟练使用多种工具来解决各种数据科学难题。每一章都按照这样的顺序组织内容：先给出一些引人入胜的示例，以便你可以整体了解这一章的内容，然后再深入细节。本书的每一节都配有习题，以帮助你实践所学到的知识。

本书适合 R 数据科学家阅读。

◆ 著　[新西兰] 哈德利•威克姆
　[美] 加勒特•格罗勒芒德
译　陈光欣
责任编辑　朱　巍
责任印制　周昇亮

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷

◆ 开本：800×1000　1/16
印张：23
字数：544千字　2018年 8 月第 1 版
印数：1 – 5 000册　2018年 8 月北京第 1 次印刷

著作权合同登记号　图字：01-2018-3442号

定价：139.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分 数据处理

第三部分　编程

第四部分 模型

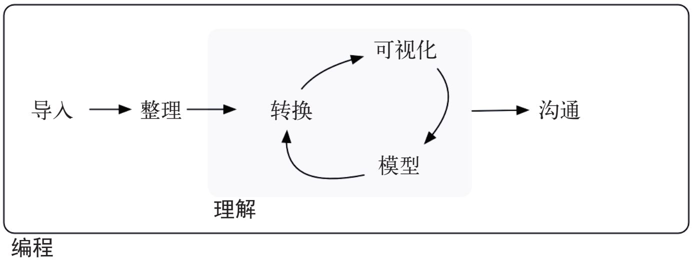
第五部分　沟通

前言

数据科学是一门激动人心的学科，它可以将原始数据转化为认识、见解和知识。本书的目标是帮助你学习使用 R 语言中最重要的数据科学工具。读完本书后，你将掌握 R 语言的精华，并能够熟练使用多种工具来解决各种数据科学难题。

你将学到什么

数据科学是一个极其广阔的领域，仅靠一本书是不可能登堂入室的。本书的目标是教会你使用最重要的数据科学工具。在一个典型的数据科学项目中，需要的工具模型大体如下图所示。



首先，你必须将数据**导入** R。这实际上就是读取保存在文件、数据库或 Web API 中的数据，再加载到 R 的数据框中。如果不能将数据导入 R，那么数据科学就根本无从谈起。

导入数据后，就应该对数据进行**整理**。数据整理就是将数据保存为一致的形式，以满足其所在数据集在语义上的要求。简而言之，如果数据是整洁的，那么每列都是一个变量，每行都是一个观测。整洁的数据非常重要，因为一致的数据结构可以让你将工作重点放在与数据有关的问题上，而不用再费尽心思地将数据转换为各种形式以适应不同的函数。

一旦拥有了整洁的数据，通常下一步就是对数据进行**转换**。数据转换包括选取出感兴趣的观测（如居住在某个城市里的所有人，或者去年的所有数据）、使用现有变量创建新变量（如根据距离和时间计算出速度），以及计算一些摘要统计量（如计数或均值）。数据整理

和数据转换统称为数据处理。

一旦使用需要的变量完成了数据整理，那么生成知识的方式主要有两种：可视化与建模。这两种方式各有利弊，相辅相成。因此，所有实际的数据分析过程都要在这两种方式间多次重复。

可视化本质上是人类活动。良好的可视化会让你发现意料之外的现象，或对数据提出新的问题。你还可以从良好的可视化中意识到自己提出了错误的问题，或者需要收集不同的数据。可视化能够带给你惊喜，但不要期望过高，因为毕竟还是需要人来对其进行解释。

模型是弥补可视化缺点的一种工具。如果已经将问题定义得足够清晰，那么你就可以使用一个模型来回答问题。因为模型本质上是一种数学工具或计算工具，所以它们的扩展性一般非常好。即使扩展性出现问题，购买更多计算机也比雇用更多聪明的人便宜！但是每个模型都有前提假设，而且模型本身不会对自己的前提假设提出疑问，这就意味着模型本质上不能给你带来惊喜。

数据科学的最后一个步骤就是**沟通**。对于任何数据分析项目来说，沟通绝对是一个极其重要的环节。如果不能与他人交流分析结果，那么不管模型和可视化让你对数据理解得多么透彻，这都是没有任何实际意义的。

围绕在这些技能之外的是**编程**。编程是贯穿数据科学项目各个环节的一项技能。数据科学家不一定是编程专家，但掌握更多的编程技能总是有好处的，因为这样你就能够对日常任务进行自动处理，并且非常轻松地解决新的问题。

你将在所有的数据科学项目中用到这些工具，但对于多数项目来说，这些工具还不够。这大致符合 80/20 定律：你可以使用从本书中学到的工具来解决每个项目中 80% 的问题，但你还需要其他工具来解决其余 20% 的问题。我们将在本书中为你提供资源，让你能够学到更多的技能。

本书的组织结构

前面对数据科学工具的描述大致是按照我们在分析中使用它们的顺序来组织的（尽管你肯定会多次使用它们）。然而，根据我们的经验，这并不是学习它们的最佳方式，具体原因如下。

- 从数据导入和数据整理开始学习并不是最佳方式，因为对于导入和整理数据来说，80% 的时间是乏味和无聊的，其余 20% 的时间则是诡异和令人沮丧的。在学习一项新技术时，这绝对是一个糟糕的开始！相反，我们将从数据可视化和数据转换开始，此时的数据已经导入并且是整理完毕的。这样一来，当导入和整理自己的数据时，你就会始终保持高昂的斗志，因为你知道这种痛苦终有回报。
- 有些主题最好结合其他工具来解释。例如，如果你已经了解可视化、数据整理和编程，那么我们认为你会更容易理解模型是如何工作的。
- 编程工具本身不一定很有趣，但它们确实可以帮助你解决更多非常困难的问题。在本书的中间部分，我们会介绍一些编程工具，它们可以与数据科学工具结合起来以解决非常有趣的建模问题。

我们尽量在每一章中使用同一种模式：先给出一些引人入胜的示例，以便你大体了解这一章的内容，然后再深入细节。本书的每一节都配有习题，以帮助你实践所学到的知识。虽然跳过这些习题是个非常有诱惑力的想法，但使用真实问题进行练习绝对是最好的学习方式。

本书未包含的内容

本书并未涉及一些重要主题。我们深信，重要的是将注意力坚定地集中在最基本的内容上，这样你就可以尽快入门并开展实际工作。这也就是说，本书不会涵盖每一个重要主题。

大数据

本书主要讨论那些小规模的、能够驻留在内存中的数据集。这是开始学习数据科学的正确方式，因为只有处理过小数据集，你才能处理大数据集。你从本书中学到的工具可以轻松地处理几百兆字节的数据，处理 1~2 GB 的数据也不会有什么大问题。如果你的日常工作是处理更大的数据（如 10~100 GB），那么你应该更多地学习一下 data.table（https://github.com/Rdatatable/data.table）。本书不会介绍 data.table，因为它的界面太过简洁，几乎没有语言提示，这使得学习起来很困难。但是如果你需要处理大数据，为了获得性能上的回报，多付出一些努力来学习它还是值得的。

如果你的数据比这还大，那么就需要仔细思考一下了，这个大数据问题是否其实就是一个小数据问题。虽然整体数据非常大，但回答特定问题所需要的数据通常较小。你可以找出一个子集、子样本或者摘要数据，该数据既适合在内存中处理，又可以回答你感兴趣的问题。此时的挑战就是如何找到合适的小数据，这通常需要多次迭代。

另外一种可能是，你的大数据问题实际上就是大量的小数据问题。每一个问题都可以在内存中处理，但你有数百万个这样的问题。举例来说，假设你想为数据集中的每个人都拟合一个模型。如果只有 10 人或 100 人，那这是小菜一碟，但如果有 100 万人，情况就完全不同了。好在每个问题都是独立于其他问题的（这种情况有时称为高度并行，embarrassingly parallel），因此你只需要一个可以将不同数据集发送到不同计算机上进行处理的系统（如 Hadoop 或 Spark）即可。如果已经知道如何使用本书中介绍的工具来解决独立子集的问题，那么你就可以学习一下新的工具（比如 sparklyr、rhipe 和 ddr）来解决整个数据集的问题。

Python、Julia以及类似的语言

在本书中，你不会学到有关 Python、Julia 以及其他用于数据科学的编程语言的任何内容。这并不是因为我们认为这些工具不好，它们都很不错！实际上，多数数据科学团队都会使用多种语言，至少会同时使用 R 和 Python。

但是，我们认为最好每次只学习并精通一种工具。如果你潜心研究一种工具，那么会比同时泛泛地学习多个工具掌握得更快。这并不是说你只应该精通一种工具，而是说每次专注于一件事情时，通常你会进步得更快。在整个职业生涯中，你都应该努力学习新事物，但是一定要在充分理解原有知识后，再去学习感兴趣的新知识。

我们认为 R 是你开始数据科学旅程的一个非常好的起点，因为它从根本上说就是一种用来支持数据科学的环境。R 不仅仅是一门编程语言，它还是进行数据科学工作的一种交互式环境。为了支持交互性，R 比多数同类语言灵活得多。虽然会导致一些缺点，但这种灵活性的一大好处是，可以非常容易地为数据科学过程中的某些环节量身定制语法。这些微型语言有助于你从数据科学家的角度来思考问题，还可以在你的大脑和计算机之间建立流畅的交流方式。

非矩形数据

本书仅关注矩形数据。矩形数据是值的集合，集合中的每个值都与一个变量和一个观测相关。很多数据集天然地不符合这种规范，比如图像、声音、树结构和文本。但是矩形数据框架在科技与工业领域是非常普遍的。我们认为它是开始数据科学旅途的一个非常好的起点。

假设验证

数据分析可以分为两类：假设生成和假设验证（有时称为验证性分析）。无须掩饰，本书的重点就在于假设生成，或者说是数据探索。我们将对数据进行深入研究，并结合专业知识生成多种有趣的假设来帮助你对数据的行为方式作出解释。你可以对这些假设进行非正式的评估，凭借自己的怀疑精神从多个方面向数据发起挑战。

假设验证与假设生成是互补的。假设验证比较困难，原因如下。

- 你需要一个精确的数学模型来生成可证伪的预测，这通常需要深厚的统计学修养。
- 为了验证假设，每个观测只能使用一次。一旦使用观测的次数超过了一次，那么就回到了探索性分析。这意味着，若要进行假设验证，你需要“预先注册”（事先拟定好）自己的分析计划，而且看到数据后也不能改变计划。在本书的第四部分中，我们将讨论一些相关的策略，你可以使用它们让假设验证变得更容易。

经常有人认为建模是用来进行假设验证的工具，而可视化是用来进行假设生成的工具。这种简单的二分法是错误的：模型经常用于数据探索；只需稍作处理，可视化也可以用来进行假设验证。核心区别在于你使用每个观测的频率：如果只用一次，那么就是假设验证；如果多于一次，那么就是数据探索。

准备工作

为了最有效地利用本书，我们对你的知识结构做了一些假设。你应该具有一定的数学基础，如果有一些编程经验也会有所帮助。如果从来没有编写过程序，那么你应该学习一下 Garrett 所著的《R 语言入门与实践》[1]，它可以作为本书的有益补充。

为了运行本书中的代码，你需要 4 个工具：R、RStudio、一个称为 tidyverse 的 R 包集合，以及另外几个 R 包。包是可重用 R 代码的基本单位，它们包括可重用的函数、描述函数使用方法的文档以及示例数据。

注 1：有关本书的详细信息，请参见图灵社区：http://www.ituring.com.cn/book/1540。

——编者注

R

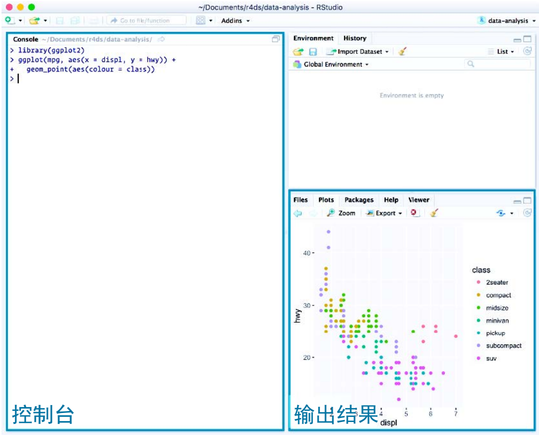
可以在 CRAN（comprehensive R archive network）下载 R。CRAN 由分布在世界各地的很多镜像服务器组成，用于分发 R 和 R 包。不要尝试选择离你近的服务器，而应该使用云镜像：https://cloud.r-project.org，它会自动找出离你最近的服务器。

R 的主版本一年发布一次，每年也会发布两三个次版本，因此你应该定期更新。更新 R 有一点麻烦，特别是更新主版本会要求你重新安装所有的 R 包，但是如果不更新的话，麻烦会更多。

RStudio

RStudio 是用于 R 编程的一种集成开发环境（integrated development environment，IDE）。你可以从 http://www.rstudio.com/download 下载并安装。RStudio 每年会更新多次。当有新版本时，RStudio 会进行通知。应该定期更新，这样你就可以使用其最新、最强大的功能。为了运行本书中的代码，请确认安装了 RStudio 1.0.0。

启动 RStudio 后，你会看到界面有以下两个关键区域。



现在，你只要知道可以在控制台窗格中输入 R 代码，然后按回车键运行就够了。在学习本书的过程中，你会学到 RStudio 的更多使用方法。

tidyverse

你还需要安装一些 R 包。R **包**是函数、数据和文档的集合，是对 R 基础功能的扩展。只有学会如何使用 R 包，才能真正掌握 R 语言的精华。你在本书中学习的大多数 R 包都是 tidyverse 的一部分。tidyverse 中的 R 包有着同样的数据处理与编程理念，它们的设计从根本上就是为了协同工作。

你可以用一行代码完整地安装 tidyverse：

```
install.packages("tidyverse")
```

在计算机上启动 RStudio 并在控制台中输入这行代码，然后按回车键来运行。R 会从 CRAN 下载这个包并将其安装在你的计算机上。如果安装有问题，请先确认你连接了互联网，再确认 https://cloud.r-project.org 没有被你的防火墙或代理服务器阻拦。

如果没有使用 `library()` 函数加载 R 包，那么你就不能使用其中的函数、对象和帮助文件。一旦 R 包安装完成，你就可以使用 `library()` 函数进行加载：

```
library(tidyverse)
#> Loading tidyverse: ggplot2
#> Loading tidyverse: tibble
#> Loading tidyverse: readr
#> Loading tidyverse: purrr
#> Loading tidyverse: dplyr
#> Conflicts with tidy packages --------------------------------
#> filter(): dplyr, stats
#> lag():    dplyr, stats
```

以上结果表明，tidyverse 正在加载 R 包 ggplot2、tibble、readr、purrr 和 dplyr。这些包被视为 tidyverse 的**核心**，因为几乎在所有的分析中都会用到它们。

tidyverse 中的包修改得相当频繁。你可以通过运行 `tidyverse_update()` 函数来检查是否有更新，并选择是否进行更新。

其他包

还有很多优秀的 R 包没有包含在 tidyverse 中，这是因为它们解决的是其他领域中的问题，或者它们遵循的是另外一套基本设计原则。它们不分优劣，只是不同而已。换句话说，与 tidyverse 互补的不是 messyverse，而是其他所有相互关联的 R 包。在使用 R 完成越来越多的数据科学项目的过程中，你会不断发现新的包，也会不断更新对数据的认识。

运行R代码

前面展示了运行 R 代码的几个示例。本书以如下方式展示 R 代码：

```
1 + 2
#> [1] 3
```

如果在本地控制台中运行同样的代码，将得到如下结果：

```
> 1 + 2
[1] 3
```

这里有两处主要区别。在控制台中，需要在 `>`（**提示符**）后面输入代码，但本书不显示提示符。书中的输出结果被 `#>` 注释掉了，但是控制台中的输出结果则直接显示在代码后面。这样一来，如果你阅读的是本书电子版，你就可以轻松地将代码从书中复制到控制台。

全书使用一致的规则来表示代码。

- 函数与代码的字体相同，并且其后有圆括号，如 `sum()` 或 `mean()`。
- 其他 R 对象（比如数据或函数参数）也使用代码字体，但其后没有圆括号，如 `flights` 或 `x`。

- 如果想要明确指出对象来自于哪个 R 包，那么我们会在包的名称后面加两个冒号，如 `dplyr::mutate()` 或 `nycflights13::flights`；R 代码也支持这种形式。

获取帮助及更多学习资源

本书并非知识孤岛，单单利用一种资源是无法精通 R 语言的。当开始将本书介绍的技术应用于自己的数据时，你很快就会发现本书并未解答所有问题。本节将介绍几个获取帮助的小技巧，以帮助你持续地学习。

如果遇到问题，首先应该求助于 Google。通常来说，在查询内容时加上一个“R”，就足以得到与 R 相关的结果。如果查不到有用的结果，这意味着目前还没有特定的 R 解决方案。Google 特别适合查询错误消息。如果收到一条错误消息，但根本不知道其含义，那就用 Google 搜索一下吧！很可能有人遇到过这种错误，而答案就在网上。（如果错误消息不是英文，可以运行 `Sys.setenv(LANGUAGE = "en")`，接着重新运行代码；使用英文错误消息进行查询更可能获得帮助。）

如果 Google 没有奏效，那么可以试试 Stack Overflow。先花点时间搜索一下现成的答案；使用 `[R]` 可以将搜索范围限定在与 R 相关的问题和答案中。如果没有发现任何有用的内容，那么就准备一个最简单的可重现实例，即 **reprex**。良好的 reprex 让你更容易从他人那里获得帮助，而且在准备 reprex 时，你往往自己就能发现问题所在。

reprex 的准备工作应该包括 3 项内容：所需 R 包、数据和代码。

- 应该在脚本开头就加载 **R 包**，这样就会很容易知道 reprex 都需要哪些 R 包。应该趁机检查自己是否使用了每个 R 包的最新版本；你可能会发现，当安装了某个 R 包的最新版本后，问题就解决了。对于 tidyverse 中的 R 包来说，检查版本的最简单方式是运行 `tidyverse_update()` 函数。
- 在 reprex 中包含数据的最简单方法是使用 `dput()` 函数生成重建数据的 R 代码。例如，要想在 R 中重建 `mtcars` 数据集，可以遵循以下步骤：

  (1) 在 R 中运行 `dput(mtcars)`；

  (2) 复制输出结果；

  (3) 在可重现脚本中输入 `mtcars <-`，然后粘贴输出结果。

  应该努力找出依然能够反映问题的数据最小子集。

- 花一点时间确保别人可以轻松理解你的**代码**：
  - 确保使用了空格，并且变量名简明扼要；
  - 用注释来说明你的问题所在；
  - 尽最大努力去除所有与问题不相关的内容。

  代码越短，越容易理解，问题也就越容易解决。

启动一个新的 R 会话，将你的脚本复制并粘贴进去，检查 reprex 是否已经准备完毕。

你还应该花些时间来防患于未然。每天花一点时间学习 R，长远来看你将获得丰厚的回

报。可以在 RStudio 博客上关注 Hadley、Garrett 和其他 RStudio 开发人员的动态。我们会在博客上发布有关新 R 包、新 IDE 功能和面授课程的一些公告。你还可以在 Twitter 上关注 Hadley（@hadleywickham）和 Garrett（@statgarrett），也可以关注 @rstudiotips 来了解 RStudio 的新功能。

为了更好地掌握 R 社区的最新动态，我们建议你关注 R-bloggers 这个网站，该网站汇集了世界各地 500 多个关于 R 的博客。如果你是活跃的 Twitter 用户，可以关注 #rstats 这个主题标签。Twitter 是 Hadley 跟踪 R 社区最新发展的一个关键工具。

致谢

本书不仅是 Hadley 和 Garrett 的作品，还是我们与 R 社区很多用户（面对面和线上）对话的结果。我们要特别感谢一些人，因为他们花费了很多时间来回答我们的问题，并帮助我们更加深刻地理解了数据科学。

- 感谢 Jenny Bryan 和 Lionel Henry 就列表和列表列的使用与我们进行了多次有益的讨论。
- 感谢 Jenny Bryan 允许我们改编其文章“R basics, workspace and working directory, RStudio projects”，进而形成了本书关于工作流的 3 章内容。
- 感谢 Genevera Allen 与我们讨论模型、建模、统计学习前景以及假设生成和假设验证的区别。
- 感谢谢益辉为 R 包 bookdown 所做的工作，同时还要感谢他不断满足我们的功能需求。
- 感谢 Bill Behrman 仔细通读了全书，并在其斯坦福数据科学课堂中试用了本书。
- 感谢使用 #rstats 主题标签的所有 Twitter 用户，他们审阅了本书全部章节的草稿，并提供了大量有用的反馈。
- 感谢 Tal Galili 扩展了其 R 包 dendextend 以支持与聚类相关的一节，虽然最终稿中并未包含这项内容。

本书是以开源方式写成的，很多人提出了修改意见并帮助改正了各种小问题。特别感谢所有通过 GitHub 为本书做出贡献的人们（按字母顺序排列）：adi pradhan、Ahmed ElGabbas、Ajay Deonarine、@Alex、Andrew Landgraf、@batpigandme、@behrman、Ben Marwick、Bill Behrman、Brandon Greenwell、Brett Klamer、Christian G. Warden、Christian Mongeau、Colin Gillespie、Cooper Morris、Curtis Alexander、Daniel Gromer、David Clark、Derwin McGeary、Devin Pastoor、Dylan Cashman、Earl Brown、Eric Watt、Etienne B. Racine、Flemming Villalona、Gregory Jefferis、@harrismcgehee、Hengni Cai、Ian Lyttle、Ian Sealy、Jakub Nowosad、Jennifer (Jenny) Bryan、@jennybc、Jeroen Janssens、Jim Hester、@jjchern、Joanne Jang、John Sears、Jon Calder、Jonathan Page、@jonathanflint、Julia Stewart Lowndes、Julian During、Justinas Petuchovas、Kara Woo、@kdpsingh、Kenny Darrell、Kirill Sevastyanenko、@koalabearski、@KyleHumphrey、Lawrence Wu、Matthew Sedaghatfar、Mine Cetinkaya-Rundel、@MJMarshall、Mustafa Ascha、@nate-d-olson、Nelson Areal、Nick Clark、@nickelas、@nwaff、@OaCantona、Patrick Kennedy、Peter Hurford、Rademeyer Vermaak、Radu Grosu、@rlzijdeman、Robert Schuessler、@robinlovelace、@robinsones、S’busiso Mkhondwane、@seamus-mckinsey、@seanpwilliams、Shannon Ellis、@shoili、@sibusiso16、@spirgel、Steve Mortimer、@svenski、Terence Teo、Thomas Klebel、TJ Mahr、Tom Prior、Will Beasley，以及谢益辉。

生成环境

本书的生成环境如下所示。

```
devtools::session_info(c("tidyverse"))
#> Session info -------------------------------------------------
#>  setting  value
#>  version  R version 3.3.1 (2016-06-21)
#>  system   x86_64, darwin13.4.0
#>  ui       X11
#>  language (EN)
#>  collate  en_US.UTF-8
#>  tz       America/Los_Angeles
#>  date     2016-10-10
#> Packages -----------------------------------------------------
#>  package      * version    date       source
#>  assertthat     0.1        2013-12-06 CRAN (R 3.3.0)
#>  BH             1.60.0-2   2016-05-07 CRAN (R 3.3.0)
#>  broom          0.4.1      2016-06-24 CRAN (R 3.3.0)
#>  colorspace     1.2-6      2015-03-11 CRAN (R 3.3.0)
#>  curl           2.1        2016-09-22 CRAN (R 3.3.0)
#>  DBI            0.5-1      2016-09-10 CRAN (R 3.3.0)
#>  dichromat      2.0-0      2013-01-24 CRAN (R 3.3.0)
#>  digest         0.6.10     2016-08-02 CRAN (R 3.3.0)
#>  dplyr        * 0.5.0      2016-06-24 CRAN (R 3.3.0)
#>  forcats        0.1.1      2016-09-16 CRAN (R 3.3.0)
#>  foreign        0.8-67     2016-09-13 CRAN (R 3.3.0)
#>  ggplot2      * 2.1.0.9001 2016-10-06 local
#>  gtable         0.2.0      2016-02-26 CRAN (R 3.3.0)
#>  haven          1.0.0      2016-09-30 local
#>  hms            0.2-1      2016-07-28 CRAN (R 3.3.1)
#>  httr           1.2.1      2016-07-03 cran (@1.2.1)
#>  jsonlite       1.1        2016-09-14 CRAN (R 3.3.0)
#>  labeling       0.3        2014-08-23 CRAN (R 3.3.0)
#>  lattice        0.20-34    2016-09-06 CRAN (R 3.3.0)
#>  lazyeval       0.2.0      2016-06-12 CRAN (R 3.3.0)
#>  lubridate      1.6.0      2016-09-13 CRAN (R 3.3.0)
#>  magrittr       1.5        2014-11-22 CRAN (R 3.3.0)
#>  MASS           7.3-45     2016-04-21 CRAN (R 3.3.1)
#>  mime           0.5        2016-07-07 cran (@0.5)
#>  mnormt         1.5-4      2016-03-09 CRAN (R 3.3.0)
#>  modelr         0.1.0      2016-08-31 CRAN (R 3.3.0)
#>  munsell        0.4.3      2016-02-13 CRAN (R 3.3.0)
#>  nlme           3.1-128    2016-05-10 CRAN (R 3.3.1)
#>  openssl        0.9.4      2016-05-25 cran (@0.9.4)
#>  plyr           1.8.4      2016-06-08 cran (@1.8.4)
#>  psych          1.6.9      2016-09-17 CRAN (R 3.3.0)
#>  purrr        * 0.2.2      2016-06-18 CRAN (R 3.3.0)
#>  R6             2.1.3      2016-08-19 CRAN (R 3.3.0)
#>  RColorBrewer   1.1-2      2014-12-07 CRAN (R 3.3.0)
#>  Rcpp           0.12.7     2016-09-05 CRAN (R 3.3.0)
#>  readr        * 1.0.0      2016-08-03 CRAN (R 3.3.0)
#>  readxl         0.1.1      2016-03-28 CRAN (R 3.3.0)
#>  reshape2       1.4.1      2014-12-06 CRAN (R 3.3.0)
#>  rvest          0.3.2      2016-06-17 CRAN (R 3.3.0)
```

```
#>  scales        0.4.0.9003 2016-10-06 local
#>  selectr       0.3-0      2016-08-30 CRAN (R 3.3.0)
#>  stringi       1.1.2      2016-10-01 CRAN (R 3.3.1)
#>  stringr       1.1.0      2016-08-19 cran (@1.1.0)
#>  tibble      * 1.2        2016-08-26 CRAN (R 3.3.0)
#>  tidyverse   * 1.0.0      2016-09-09 CRAN (R 3.3.0)
#>  xml2          1.0.0.9001 2016-09-30 local
```

排版约定

本书使用了下列排版约定。

- **黑体**
  表示新术语和重点强调的内容。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 等宽斜体（*`constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。

使用代码示例

本书源代码可以从图灵社区本书页面免费下载：http://www.ituring.com.cn/book/2113。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O’Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如：“*R for Data Science* by Hadley Wickham and Garrett Grolemund (O’Reilly). Copyright 2017 Garrett Grolemund, Hadley Wickham, 978-1-491-91039-9.”

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly Safari

Safari（原来叫 Safari Books Online）是一个会员制的培训和参考咨询平台，面向企业、政府、教育从业者和个人。

会员可以访问几千种书籍、培训视频、学习路径、互动式教程和精选播放列表，提供这些资源的出版商超过 250 家，包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology，等等。

要获得更多信息，请访问 http://oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你在上面可以找到图书的相关信息，包括勘误表、示例代码以及其他信息。

对于本书的评论和技术性问题，请发送电子邮件到：
bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

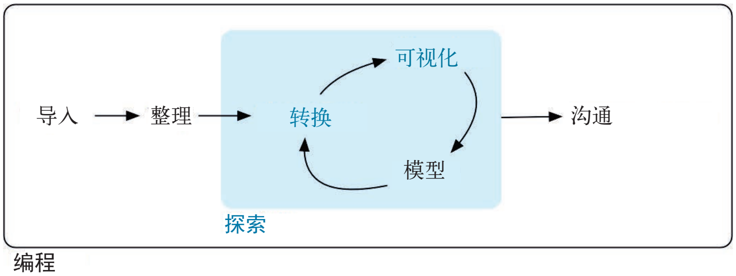
电子书

扫描如下二维码，即可购买本书电子版。

第一部分

探索

本书第一部分的目的是让你尽快掌握**数据探索**的基本工具。数据探索是一门艺术，它可以审视数据，快速生成假设并进行检验，接着重复、重复、再重复。数据探索的目的是生成多个有分析价值的线索，以供后续进行更深入的研究。



你将在本部分中学习一些非常有用的工具，它们的效果立竿见影。

- 可视化是开始 R 编程的一个非常好的起点，因为其回报非常明确：你可以做出样式优雅且信息丰富的图形来帮助自己理解数据。在第 1 章中，你将深入钻研数据可视化，学习 ggplot2 图形的基本结构以及将数据转换为图形的强大技术。
- 只进行可视化通常是不够的，因此你将在第 3 章中学习一些非常重要的操作，其中包括选取重要变量、筛选关键观测、创建新变量，以及计算摘要统计量。
- 最后，在第 5 章中，你将利用数据可视化技术和数据转换技术，结合你的好奇心和怀疑精神，对数据提出有趣的问题并试图找到答案。

建模是数据探索过程中非常重要的环节，但你现在还没有掌握有效学习和应用模型的技能。一旦你掌握了更多的数据处理工具和编程工具，我们将在第四部分继续讨论建模技术。

在讲授数据探索工具的 3 章间，我们穿插了介绍 R 工作流的 3 章内容。在第 2 章、第 4 章和第 6 章中，你将学习编写和组织 R 代码的最佳实践。从长远来看，这会为你的成功打下坚实的基础，因为这几章介绍的工具可以让你井井有条地处理实际项目。

第 1 章

使用ggplot2进行数据可视化

1.1 简介

简单的图形对数据分析者的启示比任何其他方法都要多。

——John Tukey

本章将教你如何使用 ggplot2 进行数据可视化。R 有好几种绘图工具，但 ggplot2 是其中最优雅、功能最全面的一个。ggplot2 实现了**图形语法**，这是一套用来描述和构建图形的连贯性语法规则。掌握 ggplot2 后，你便可以在多个场景中使用，从而显著提高工作效率。

在开始学习之前，如果想要了解 ggplot2 背后的更多理论基础，推荐你读一读“A Layered Grammar of Graphics”。

准备工作

本章重点讨论 tidyverse 的一个核心 R 包——ggplot2。为了访问本章用到的数据集、帮助页面和函数，需要先运行以下代码来加载 tidyverse：

```
library(tidyverse)
#> Loading tidyverse: ggplot2
#> Loading tidyverse: tibble
#> Loading tidyverse: readr
#> Loading tidyverse: purrr
#> Loading tidyverse: dplyr
#> Conflicts with tidy packages --------------------------------
#> filter(): dplyr, stats
#> lag():    dplyr, stats
```

这一行代码加载了 tidyverse 的核心 R 包。在几乎所有的数据分析任务中，你都会用到这些 R 包。这行代码还会告诉你 tidyverse 中的哪些函数与基础 R 包（或者已加载的其他 R 包）中的函数有冲突。

如果运行这行代码时出现错误消息“there is no package called ‘tidyverse’”，那么你需要先安装 tidyverse，然后再运行 `library()` 函数：

```
install.packages("tidyverse")
library(tidyverse)
```

R 包只需安装一次，但每次开始新会话时都要重新加载。

如果想要明确指出某个函数（或数据集）的来源，那么可以使用特殊语法形式 `package::function()`。例如，`ggplot2::ggplot()` 明确指出了我们使用的是 ggplot2 包中的 `ggplot()` 函数。

1.2 第一步

我们使用第一张图来回答问题：大引擎汽车比小引擎汽车更耗油吗？你可能已经有了答案，但应该努力让答案更精确一些。引擎大小与燃油效率之间是什么关系？是正相关，还是负相关？是线性关系，还是非线性关系？

1.2.1 mpg数据框

你可以使用 ggplot2 包中的 `mpg` 数据框（即 `ggplot2::mpg`）来检验自己的答案。**数据框**是变量（列）和观测（行）的矩形集合。`mpg` 包含了由美国环境保护协会收集的 38 种车型的观测数据。

```
mpg
#> # A tibble: 234 × 11
#>   manufacturer model displ  year   cyl      trans   drv
#>          <chr> <chr> <dbl> <int> <int>      <chr> <chr>
#> 1         audi    a4   1.8  1999     4   auto(l5)     f
#> 2         audi    a4   1.8  1999     4 manual(m5)     f
#> 3         audi    a4   2.0  2008     4 manual(m6)     f
#> 4         audi    a4   2.0  2008     4   auto(av)     f
#> 5         audi    a4   2.8  1999     6   auto(l5)     f
#> 6         audi    a4   2.8  1999     6 manual(m5)     f
#> # ... with 228 more rows, and 4 more variables:
#> # cty <int>, hwy <int>, fl <chr>, class <chr>
```

`mpg` 中包括如下变量。

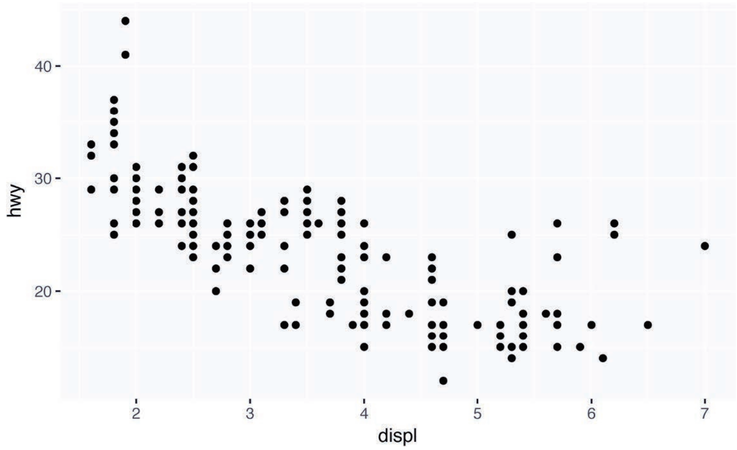
- `displ`：引擎大小，单位为升。
- `hwy`：汽车在高速公路上行驶时的燃油效率，单位为英里 / 加仑（mpg）。与燃油效率高的汽车相比，燃油效率低的汽车在行驶相同距离时要消耗更多燃油。

要想了解更多关于 `mpg` 的信息，可以使用 `?mpg` 命令打开其帮助页面。

1.2.2 创建ggplot图形

为了绘制 mpg 的图形，运行以下代码将 displ 放在 x 轴，hwy 放在 y 轴：

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```



上图显示出引擎大小（displ）和燃油效率（hwy）之间是负相关关系。换句话说，大引擎汽车更耗油。这张图是证实了你对燃油效率和引擎大小之间关系的假设，还是推翻了它？

在 ggplot2 中，你可以使用 ggplot() 函数开始绘图。ggplot() 创建了一个坐标系，你可以在它上面添加图层。ggplot() 的第一个参数是要在图中使用的数据集。ggplot(data = mpg) 会创建一张空白图，因为这张图没什么意思，所以就不在这里展示了。

向 ggplot() 中添加一个或多个图层就可以完成这张图。函数 geom_point() 向图中添加一个点层，这样就可以创建一张散点图。ggplot2 中包含了多种几何对象函数，每种函数都可以向图中添加不同类型的图层。你将在本章中学到大量的几何对象函数。

ggplot2 中的每个几何对象函数都有一个 mapping 参数。这个参数定义了如何将数据集中的变量映射为图形属性。mapping 参数总是与 aes() 函数成对出现，aes() 函数的 x 参数和 y 参数分别指定了映射到 x 轴的变量与映射到 y 轴的变量。ggplot2 在 data 参数中寻找映射变量，本例中就是 mpg。

1.2.3 绘图模板

我们将上面的代码转换为一个可重用的 ggplot2 绘图模板。要想生成一张图，将以下代码中的尖括号部分替换为数据集、几何对象函数或映射集合即可：

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

本章其余内容将向你展示如何完成并扩展这个模板，以制作出各种类型的图。我们将从 <MAPPINGS> 部分开始。

1.2.4 练习

(1) 运行 `ggplot(data = mpg)`，你会看到什么？

(2) 数据集 `mpg` 中有多少行？多少列？

(3) 变量 `drv` 的意义是什么？使用 `?mpg` 命令阅读帮助文件以找出答案。

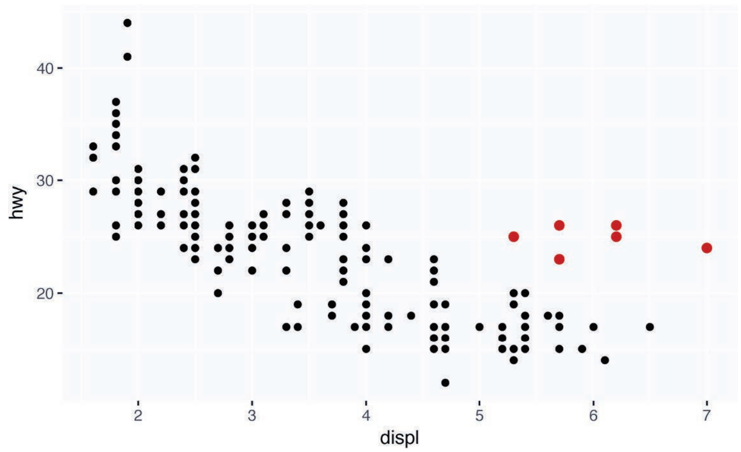
(4) 使用 `hwy` 和 `cyl` 绘制一张散点图。

(5) 如果使用 `class` 和 `drv` 绘制散点图，会发生什么情况？为什么这张图没什么用处？

1.3 图形属性映射

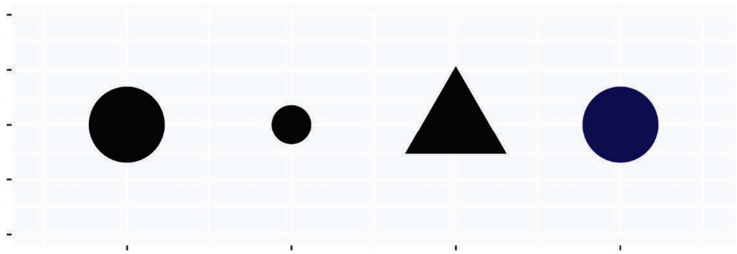
> 图片的最大价值在于促使我们发现从未预料到的事情。
>
> ——John Tukey

下图中有一组点（显示为红色）似乎位于线性趋势之外。这些汽车比预期具有更高的里程数。如何解释这种现象呢？



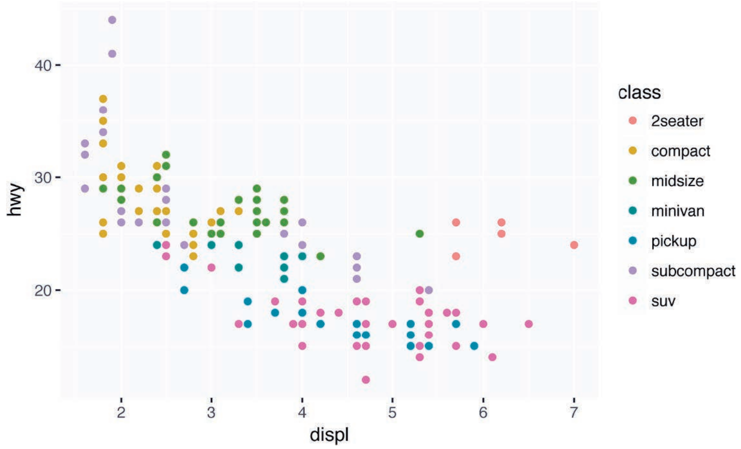
我们可以假设这些汽车是混合动力车。检验这种假设的一个方法是查看每辆汽车的 `class` 值。`mpg` 数据集中的 `class` 变量对汽车进行了分类，比如小型、中型和 SUV。如果那些离群点是混合动力车，那么它们应该分类为小型车，也可能是微型车（注意，这份数据是在混合动力车和 SUV 流行前收集的）。

可以向二维散点图中添加第三个变量，比如 `class`，方式是将它映射为**图形属性**。图形属性是图中对象的可视化属性，其中包括数据点的大小、形状和颜色。通过改变图形属性的值，可以用不同的方式来显示数据点（如下图所示）。因为已经使用“value”这个词来表示数据的值，所以下面使用“level”（水平）这个词来表示图形属性的值。我们来改变一个点的大小、形状和颜色的水平，分别让它变小、变为三角形和变为蓝色，如下所示。



通过将图中的图形属性映射为数据集中的变量，可以传达出数据的相关信息。例如，可以将点的颜色映射为变量 class，从而揭示每辆汽车的类型：

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class))
```



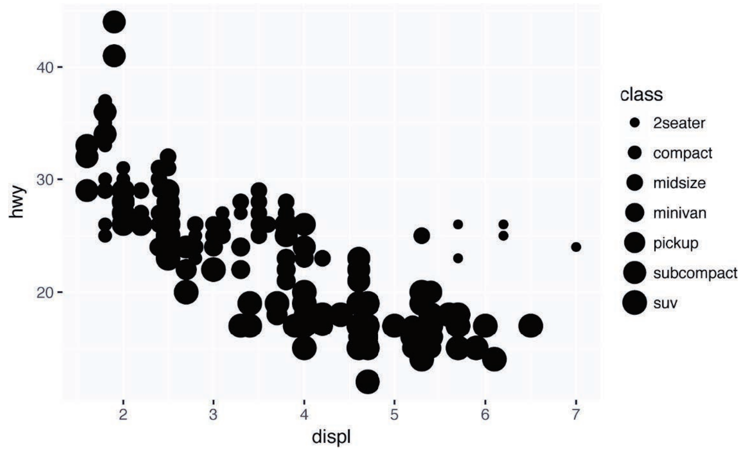
（如果你更喜欢英式英语，就像 Hadley 一样，那么可以使用 colour 代替 color。）

要想将图形属性映射为变量，需要在函数 aes() 中将图形属性名称和变量名称关联起来。ggplot2 会自动为每个变量值分配唯一的图形属性水平（本例中是唯一的颜色），这个过程称为**标度变换**。ggplot2 还会添加一个图例，以表示图形属性水平和变量值之间的对应关系。

颜色揭示出的信息是，那些离群点多数是双座汽车。这些汽车不像是混合动力车；实际上，它们是跑车！跑车像 SUV 和皮卡一样配有大引擎，但车身却类似于中型车和小型车，这样就提高了它的里程数。现在想来，这些车不会是混合动力的，因为它们具有大引擎。

在上例中，我们将 class 映射为颜色，但也可以用同样的方式将其映射为点的大小。在下面的示例中，每个点的实际大小表示其所属的类别。这里我们收到一条**警告信息**，因为将无序变量（class）映射为有序图形属性（size）可不是好主意。

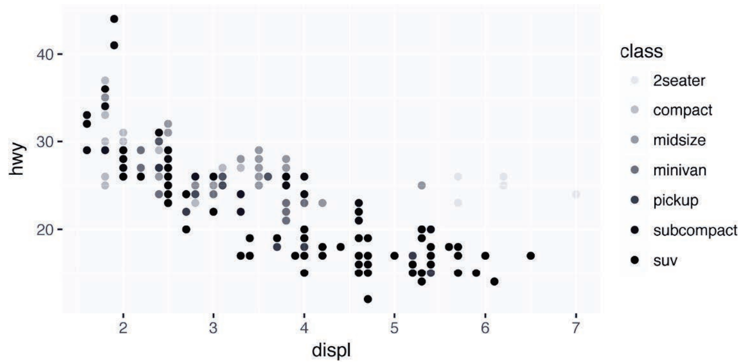
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, size = class))
#> Warning: Using size for a discrete variable is not advised.
```

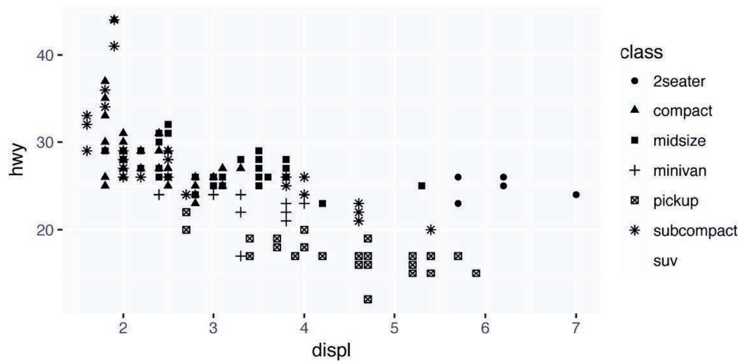


或者我们也可以将 class 映射为控制数据点透明度的 alpha 图形属性，还可以将其映射为点的形状。

```
# 上图
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, alpha = class))

# 下图
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```





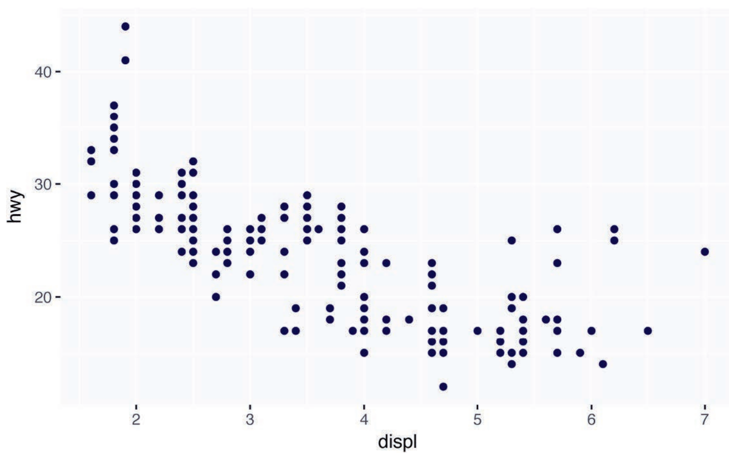
SUV 怎么了？ ggplot2 只能同时使用 6 种形状。默认情况下，当使用这种图形属性时，多出的变量值将不会出现在图中。

对你所使用的每个图形属性来说，函数 `aes()` 都可以将其名称与一个待显示变量关联起来。`aes()` 将图层中使用的每个图形属性映射集合在一起，然后传递给该图层的映射参数。这一语法强调了关于 x 和 y 的重要信息：数据点的 *x* 轴位置和 *y* 轴位置本身就是图形属性，即可以映射为变量来表示数据信息的可视化属性。

一旦映射了图形属性，ggplot2 会处理好其余的事情。它会为图形属性选择一个合适的标度，并创建图例来表示图形属性水平和变量值之间的映射关系。ggplot2 不会为 x 和 y 这两个图形属性创建图例，而会创建带有刻度标记和标签的坐标轴。坐标轴就相当于图例，可以体现出位置和变量值之间的映射关系。

还可以手动为几何对象**设置**图形属性。例如，我们可以让图中的所有点都为蓝色：

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy), color = "blue")
```



此时颜色不会传达关于变量的信息，只是改变图的外观。要想手动设置图形属性，需要按名称进行设置，将其作为几何对象函数的一个参数。这也就是说，需要在函数 `aes()` 的外部进行设置。此外，还需要为这个图形属性选择一个有意义的值。

- 颜色名称是一个字符串。
- 点的大小用毫米表示。
- 点的形状是一个数值，如图 1-1 所示。有些形状相同，比如 0、15 和 22 都是正方形。形状之间的区别在于 `color` 和 `fill` 这两个图形属性。空心形状（0~14）的边界颜色由 `color` 决定；实心形状（15~20）的填充颜色由 `color` 决定；填充形状（21~24）的边界颜色由 `color` 决定，填充颜色由 `fill` 决定。

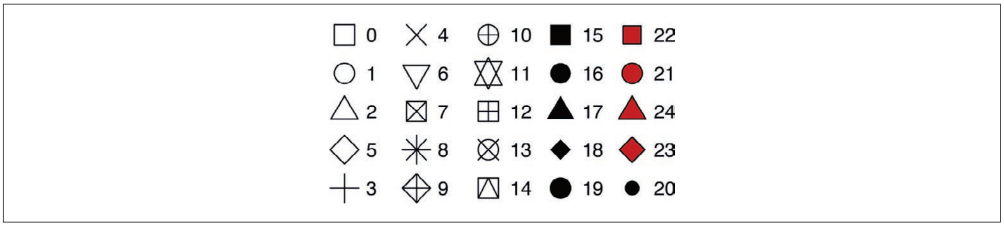
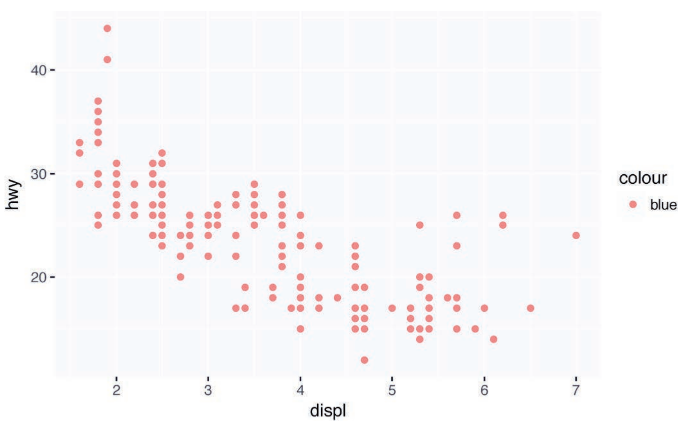


图 1-1：用数值进行标识的 R 的 25 种内置形状

练习

(1) 以下这段代码有什么错误？为什么点不是蓝色的？

```
ggplot(data = mpg) +
  geom_point(
    mapping = aes(x = displ, y = hwy, color = "blue")
  )
```



(2) mpg 中的哪些变量是分类变量？哪些变量是连续变量？（提示：输入 ?mpg 来阅读这个数据集的文档。）当调用 mpg 时，如何才能看到这些信息？

(3) 将一个连续变量映射为 color、size 和 shape。对分类变量和连续变量来说，这些图形属性的表现有什么不同？

(4) 如果将同一个变量映射为多个图形属性，会发生什么情况？

(5) stroke 这个图形属性的作用是什么？它适用于哪些形状？（提示：使用 ?geom_point 命令。）

(6) 如果将图形属性映射为非变量名对象，比如 aes(color = displ < 5)，会发生什么情况？

1.4　常见问题

当开始运行 R 代码时，你很可能会遇到问题。不用担心，每个人都会遇到问题。我已经编写 R 代码多年了，但还是每天都会写出不能正常运行的代码。

首先，将你需要运行的代码与书中的代码进行仔细对比。R 极其挑剔，即使一个字母放错了位置，也可能会造成问题。确保每个 (都有一个) 与之匹配，并且每个 " 后面都跟着另一个 "。有时运行了代码却什么也没有发生。检查一下控制台左侧：如果有一个 + 号，那么说明 R 认为你没有输入完整的表达式，正在等待你完成输入。这种情况下，按 Esc 键中止当前执行的命令就可以重新开始。

创建 ggplot2 图形时的一个常见问题是将 + 号放错了位置：+ 必须放在一行代码的末尾，而不是开头。换句话说，请确保你没有粗心地写出以下这样的代码：

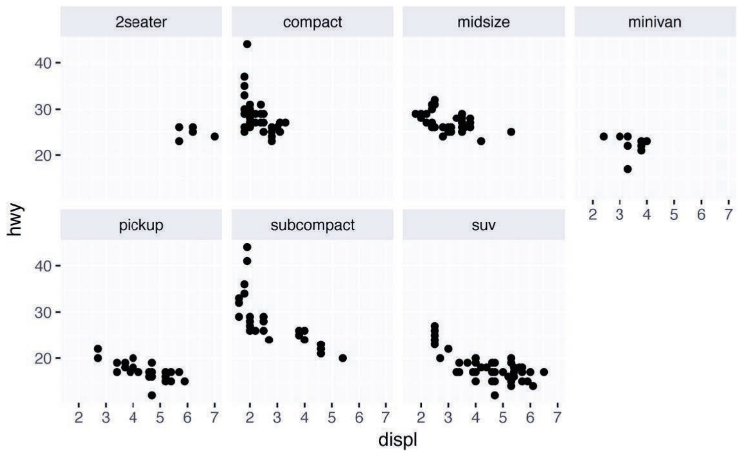
```
ggplot(data = mpg)
+ geom_point(mapping = aes(x = displ, y = hwy))
```

如果还是有问题，那么可以看看帮助页面。通过在控制台中运行 ? 函数名，或者在 RStudio 中选定函数名称后按 F1 键，你可以获得任何 R 函数的帮助信息。如果帮助页面看上去没什么用，也不要着急，你可以跳过这些帮助信息，向下找到示例部分，并查看与你的需求相匹配的代码。

如果还是没什么用，那么再仔细阅读一下错误消息。有时答案就隐藏在其中！但如果你是 R 语言新手，那么即使答案就在错误消息中，你也很难理解。另一个非常好的工具就是 Google：试着搜索一下错误消息，因为别人很可能也遇到过同样的问题，并在网上得到了答案。

1.5 分面

添加额外变量的一种方法是使用图形属性。另一种方法是将图分割成多个**分面**，即可以显示数据子集的子图。这种方法特别适合添加分类变量。

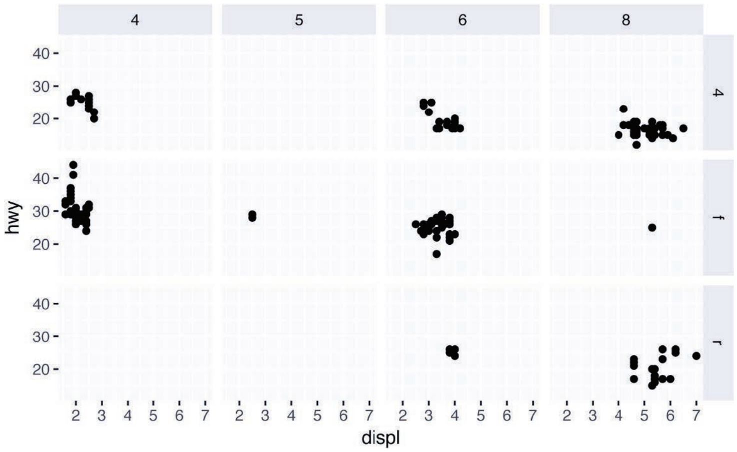


要想通过单个变量对图进行分面，可以使用函数 `facet_wrap()`。其第一个参数是一个公式，创建公式的方式是在 ~ 符号后面加一个变量名（这里所说的“公式”是 R 中的一种数据结构，不是数学意义上的公式）。传递给 `facet_wrap()` 的变量应该是离散型的。

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```

要想通过两个变量对图进行分面，需要在绘图命令中加入函数 facet_grid()。这个函数的第一个参数也是一个公式，但该公式包含由 ~ 隔开的两个变量名。

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(drv ~ cyl)
```



如果不想在行或列的维度进行分面，你可以使用 . 来代替变量名，例如 + facet_grid(. ~ cyl)。

练习

(1) 如果使用连续变量进行分面，会发生什么情况？

(2) 在使用 facet_grid(drv ~ cyl) 生成的图中，空白单元的意义是什么？它们和以下代码生成的图有什么关系？

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = drv, y = cyl))
```

(3) 以下代码会绘制出什么图？ . 的作用是什么？

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(drv ~ .)

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(. ~ cyl)
```

(4) 查看本节的第一个分面图：

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```

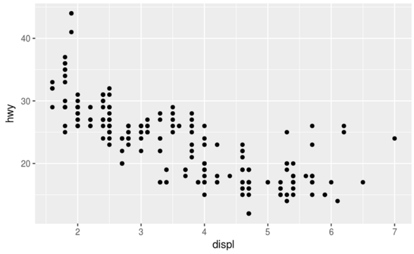
与使用图形属性相比，使用分面的优势和劣势分别是什么？如果有一个更大的数据集，你将如何权衡这两种方法的优劣？

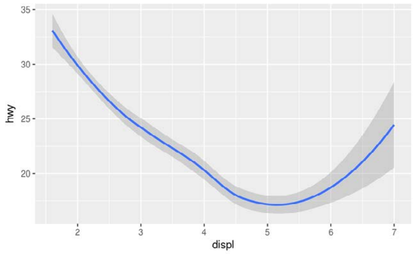
(5) 阅读 ?facet_wrap 的帮助页面。nrow 和 ncol 的功能分别是什么？还有哪些选项可以控制分面的布局？为什么函数 facet_grid() 没有变量 nrow 和 ncol？

(6) 在使用函数 facet_grid() 时，一般应该将具有更多唯一值的变量放在列上。为什么这么做呢？

1.6 几何对象

以下两张图的相似程度有多大？





两张图有同样的 x 变量和 y 变量，而且描述的是同样的数据。但这两张图并不一样，它们各自使用不同的可视化对象来表示数据。在 ggplot2 语法中，我们称它们使用了不同的**几何对象**。

几何对象是图中用来表示数据的几何图形对象。我们经常根据图中使用的几何对象类型来描述相应的图。例如，条形图使用了条形几何对象，折线图使用了直线几何对象，箱线图使用了矩形和直线几何对象。散点图打破了这种趋势，它们使用点几何对象。如上面的两幅图所示，我们可以使用不同的几何对象来表示同样的数据。左侧的图使用了点几何对象，右侧的图使用了平滑曲线几何对象，以一条平滑曲线来拟合数据。

要想改变图中的几何对象，需要修改添加在 ggplot() 函数中的几何对象函数。举例来说，要想绘制出上图，你可以使用以下代码：

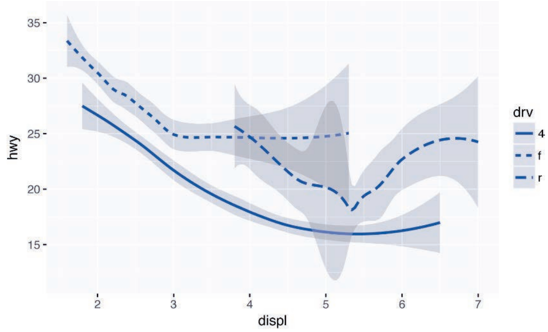
```
# 左图
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))

# 右图
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

ggplot2 中的每个几何对象函数都有一个 mapping 参数。但是，不是每种图形属性都适合每种几何对象。你可以设置点的形状，但不能设置线的“形状”，而可以设置线的线型。

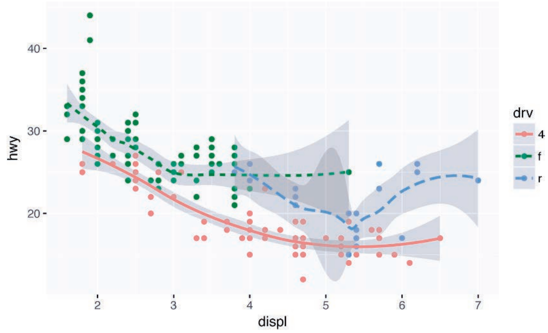
geom_smooth() 函数可以按照不同的线型绘制出不同的曲线，每条曲线对应映射到线型的变量的一个唯一值：

```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, linetype = drv))
```



根据表示汽车驱动系统的 drv 变量的值，这里的 geom_smooth() 函数分别用 3 条曲线来表示汽车。一条线表示 drv 值为 4 的所有汽车，一条线表示 drv 值为 f 的所有汽车，另一条线表示 drv 值为 r 的所有汽车。其中 4 表示四轮驱动，f 表示前轮驱动，r 表示后轮驱动。

如果你觉得这有些难以理解，我们可以将这些曲线覆盖在原始数据上，并按照 drv 值对所有的点和线进行着色，这样你就能看得更清楚一些了。



注意，我们刚才在同一张图中使用了两种几何对象，多么激动人心！稍安勿躁，我们将在下一节中学习如何在同一张图中放置多个几何对象。

ggplot2 提供了 30 多种几何对象，其扩展包甚至提供了更多（可以在 https://www.ggplot2-exts.org 查看更多样例）。如果想全面地了解这些对象，最好的方式是学习 ggplot2 速查表（参见 http://rstudio.com/cheatsheets）。如果想掌握更多关于某个几何对象的知识，那么可以使用帮助，如 ?geom_smooth。

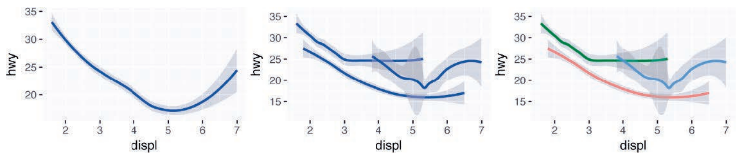
和 geom_smooth() 一样，很多几何对象函数使用单个几何对象来表示多行数据。你可以将这些几何对象的 group 图形属性设置为一个分类变量，这样 ggplot2 就会为这个分类变量的每个唯一值绘制一个独立的几何对象。实际上，只要将一个图形属性映射为一个离散变量（如上个示例中的 linetype），ggplot2 就会自动对数据进行分组来绘制多个几何对象。

这个功能非常方便，因为按照图形属性的这种分组不用添加图例，也不用为几何对象添加区分特征：

```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))

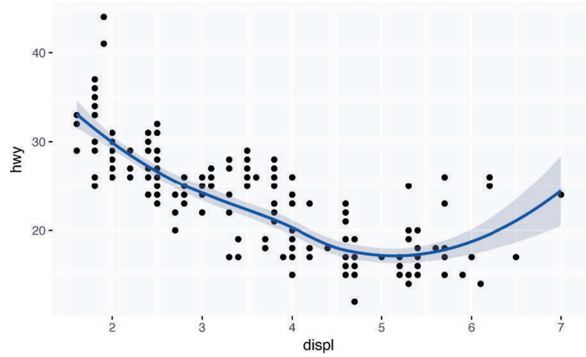
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, group = drv))

ggplot(data = mpg) +
  geom_smooth(
    mapping = aes(x = displ, y = hwy, color = drv),
    show.legend = FALSE
  )
```



要想在同一张图中显示多个几何对象，可以向 ggplot() 函数中添加多个几何对象函数：

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

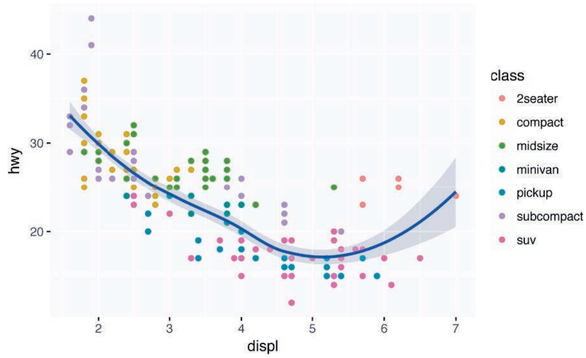


但是，这样代码就产生了一些重复。假如你想将 *y* 轴上的变量从 hwy 改成 cty，那么就要在两个地方修改这个变量，但你或许会漏掉一处。避免这种重复的方法是将一组映射传递给 ggplot() 函数。ggplot2 会将这些映射作为全局映射应用到图中的每个几何对象中。换句话说，以下代码将绘制出与上面代码同样的图：

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth()
```

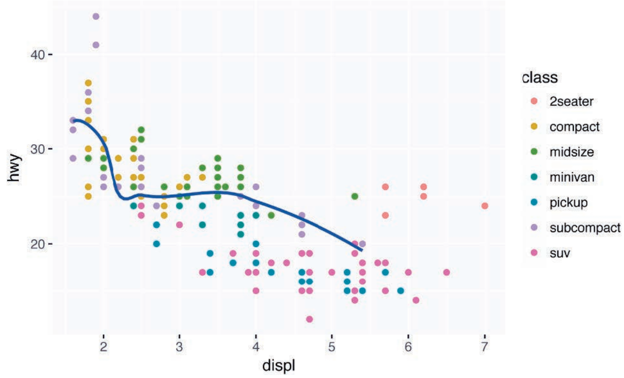
如果将映射放在几何对象函数中，那么 ggplot2 会将其看作这个图层的局部映射，它将使用这些映射扩展或覆盖全局映射，**但仅对该图层有效**。这样一来，我们就可以在不同的图层中显示不同的图形属性：

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth()
```



同理，你也可以为不同的图层指定不同的数据。下图中的平滑曲线表示的只是 mpg 数据集的一个子集，即微型车。geom_smooth() 函数中的局部数据参数覆盖了 ggplot() 函数中的全局数据参数，当然仅对这个图层有效：

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth(
    data = filter(mpg, class == "subcompact"),
    se = FALSE
  )
```



（你将在下一章中学习 filter() 函数的用法，现在只需要知道这个命令仅选取出微型车就够了。）

练习

(1) 在绘制折线图、箱线图、直方图和分区图时，应该分别使用哪种几何对象？

(2) 在脑海中运行以下代码，并预测会有何种输出。接着在 R 中运行代码，并检查你的预测是否正确。

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy, color = drv)
) +
  geom_point() +
  geom_smooth(se = FALSE)
```

(3) show.legend = FALSE 的作用是什么？删除它会发生什么情况？你觉得我为什么要在本章前面的示例中使用这句代码？

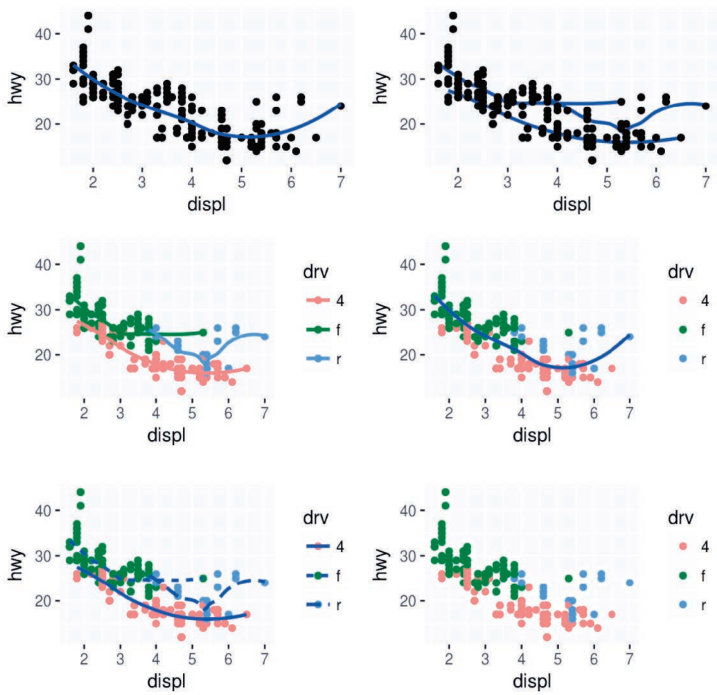
(4) geom_smooth() 函数中的 se 参数的作用是什么？

(5) 以下代码生成的两张图有什么区别吗？为什么？

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth()

ggplot() +
  geom_point(
    data = mpg,
    mapping = aes(x = displ, y = hwy)
  ) +
  geom_smooth(
    data = mpg,
    mapping = aes(x = displ, y = hwy)
  )
```

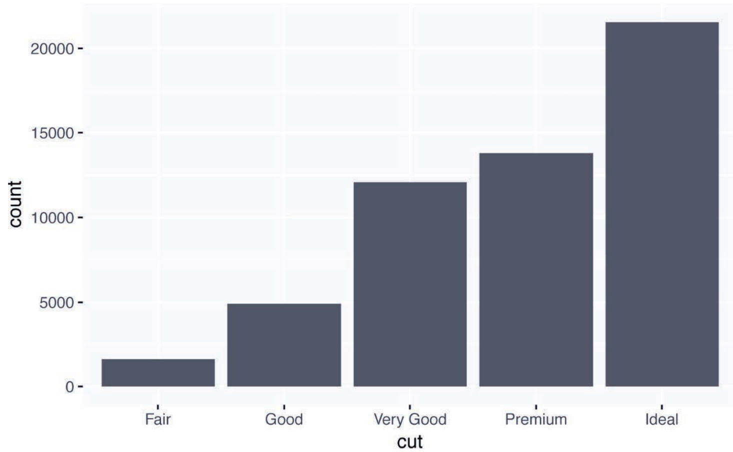
(6) 自己编写 R 代码来生成以下各图。



1.7 统计变换

接下来我们看一下条形图。条形图虽然简单，但很有意思，因为它可以揭示出图形中的一些微妙信息。我们看一下用 `geom_bar()` 函数就可以绘制的基本条形图。下面的条形图显示了 `diamonds` 数据集中按照 `cut` 变量分组的各种钻石的总数量。`diamonds` 数据集是 ggplot2 的内置数据集，包含大约 54 000 颗钻石的信息，每颗钻石具有 `price`、`carat`、`color`、`clarity` 和 `cut` 变量。条形图显示，高质量切割钻石的数量要比低质量切割钻石的数量多：

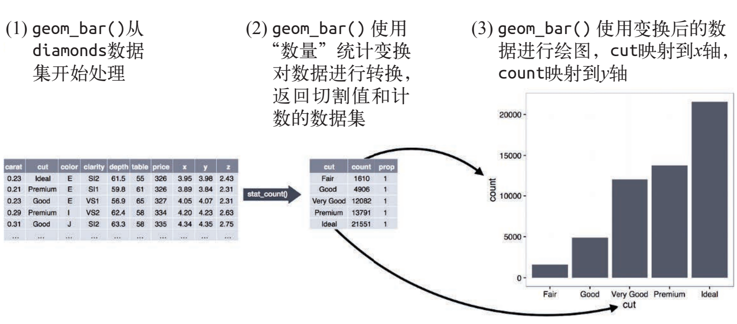
```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```



条形图 x 轴显示的是 `cut`，这是 `diamonds` 数据集中的一个变量。y 轴显示的是 `count`，但 `count` 不是 `diamonds` 中的变量！那么 `count` 来自哪里呢？很多图形绘制的是数据集的原始数据，比如散点图。另外一些图形则可以绘制那些计算出的新数据，比如条形图。

- 条形图、直方图和频率多边形图可以对数据进行分箱，然后绘制出分箱数量和落在每个分箱的数据点的数量。
- 平滑曲线会为数据拟合一个模型，然后绘制出模型预测值。
- 箱线图可以计算出数据分布的多种摘要统计量，并显示一个特殊形式的箱体。

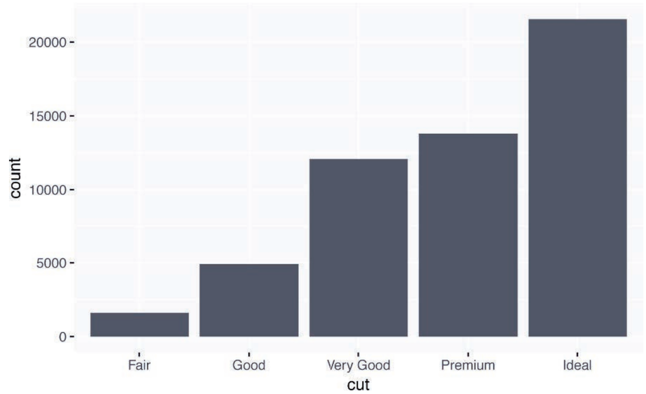
绘图时用来计算新数据的算法称为 stat（statistical transformation，**统计变换**）。下图描述了 `geom_bar()` 函数的统计变换过程。



通过查看 stat 参数的默认值，你可以知道几何对象函数使用了哪种统计变换。例如，?geom_bar 显示出 stat 的默认值是 count，这说明 geom_bar() 使用 stat_count() 函数进行统计变换。stat_count() 在文档中与 geom_bar() 位于同一页，如果继续向下看，你可以发现名为“Computed variables”的一节，它告诉我们 stat_count() 会计算出两个新变量：count 和 prop。

通常来说，几何对象函数和统计变换函数可以互换使用。例如，你可以使用 stat_count() 替换 geom_bar() 来重新生成前面那张图：

```
ggplot(data = diamonds) +
  stat_count(mapping = aes(x = cut))
```

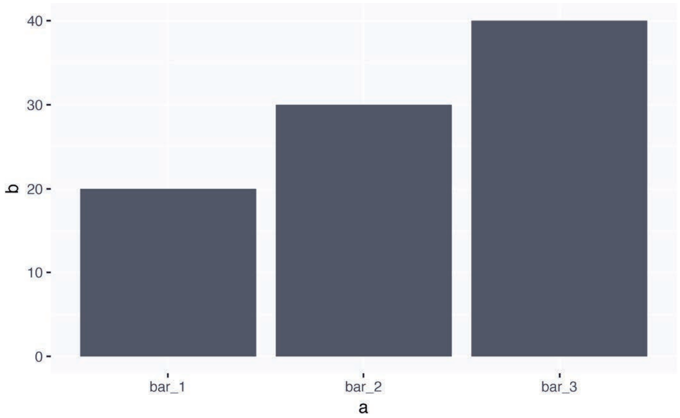


可以这样做的原因是，每个几何对象函数都有一个默认统计变换，每个统计变换函数都有一个默认几何对象。一般情况下，这意味着你在使用几何对象函数时不用担心底层的统计变换。想要显式使用某种统计变换的 3 个原因如下。

- 你可能想要覆盖默认的统计变换。在以下代码中，我们将 geom_bar() 函数的统计变换从计数（默认值）修改为标识。这样我们就可以将条形的高度映射为 y 轴变量的初始值。遗憾的是，当随意说起条形图时，人们指的可能就是这种条形图，其中条形高度已经存在于数据中，而不是像前一个图一样，条形高度由对行进行计数来生成：

```
demo <- tribble(
  ~a,      ~b,
  "bar_1", 20,
  "bar_2", 30,
  "bar_3", 40
)

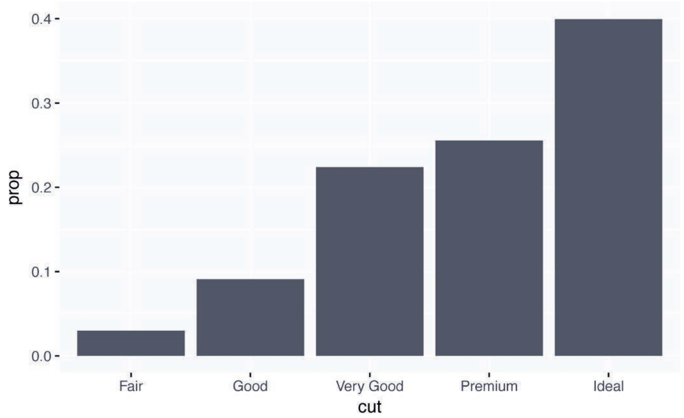
ggplot(data = demo) +
  geom_bar(
    mapping = aes(x = a, y = b), stat = "identity"
  )
```



（你还不知道 <- 和 tibble() 的含义？别担心，根据上下文就能猜出它们的含义，而且你很快就会学习它们了。）

- 你可能想要覆盖从统计变换生成的变量到图形属性的默认映射。例如，你或许想显示一张表示比例（而不是计数）的条形图：

```
ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, y = ..prop.., group = 1)
  )
```

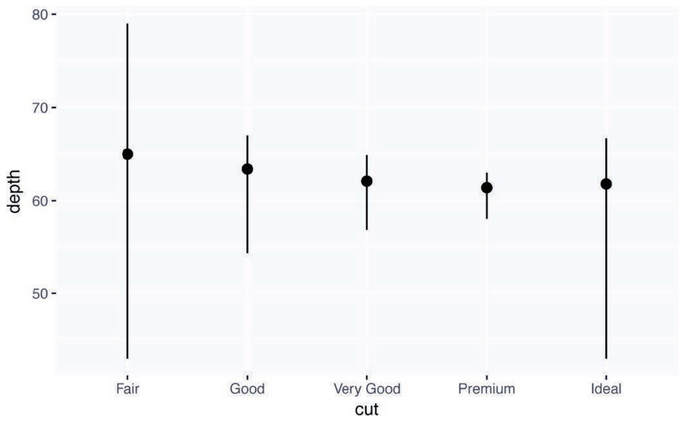


如果想要找出由统计变换计算出的变量，可以查看帮助文件中的“Computed variables”一节。

- 你可能想要在代码中强调统计变换。例如，你可以使用 stat_summary() 函数将人们的注意力吸引到你计算出的那些摘要统计量上。stat_summary() 函数为 x 的每个唯一值计算 y 值的摘要统计：

```
ggplot(data = diamonds) +
  stat_summary(
    mapping = aes(x = cut, y = depth),
```

```
    fun.ymin = min,
    fun.ymax = max,
    fun.y = median
  )
```



ggplot2 提供了 20 多个统计变换以供你使用。每个统计变换都是一个函数，因此你可以按照通用方式获得帮助，例如 `?stat_bin`。如果想要查看全部的统计变换，可以使用 ggplot2 速查表。

练习

(1) `stat_summary()` 函数的默认几何对象是什么？不使用统计变换函数的话，如何使用几何对象函数重新生成以上的图？

(2) `geom_col()` 函数的功能是什么？它和 `geom_bar()` 函数有何不同？

(3) 多数几何对象和统计变换都是成对出现的，总是配合使用。仔细阅读文档，列出所有成对的几何对象和统计变换。它们有什么共同之处？

(4) `stat_smooth()` 函数会计算出什么变量？哪些参数可以控制它的行为？

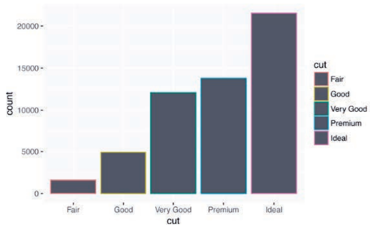
(5) 在比例条形图中，我们需要设定 `group = 1`，这是为什么呢？换句话说，以下两张图会有什么问题？

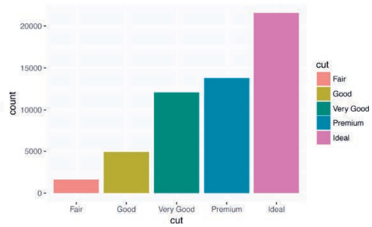
```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, y = ..prop..))
ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = color, y = ..prop..)
  )
```

1.8 位置调整

条形图还有一项神奇的功能，你可以使用 `color` 或者 `fill`（这个更有用）图形属性来为条形图上色：

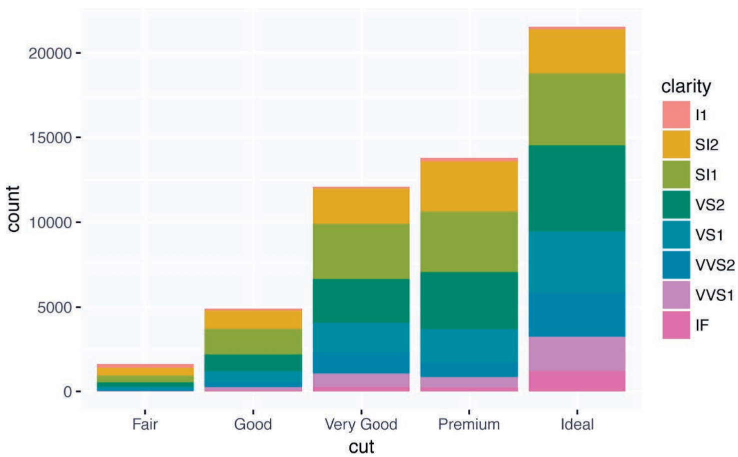
```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, color = cut))
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = cut))
```





注意，如果将 fill 图形属性映射到另一个变量（如 clarity），那么条形会自动分块堆叠起来。每个彩色矩形表示 cut 和 clarity 的一种组合。

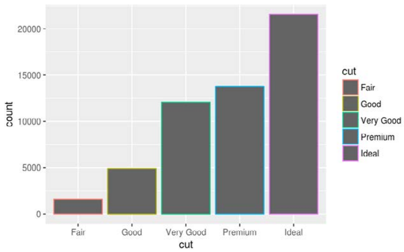
```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = clarity))
```

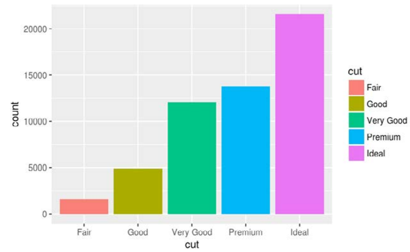


这种堆叠是由 position 参数设定的**位置调整**功能自动完成的。如果不想生成堆叠式条形图，你还可以使用以下 3 种选项之一："identity"、"fill" 和 "dodge"。

- position = "identity" 将每个对象直接显示在图中。这种方式不太适合条形图，因为条形会彼此重叠。为了让重叠部分能够显示出来，我们可以设置 alpha 参数为一个较小的数，从而使得条形略微透明；或者设定 fill = NA，让条形完全透明：

```
ggplot(
  data = diamonds,
  mapping = aes(x = cut, fill = clarity)
) +
  geom_bar(alpha = 1/5, position = "identity")
ggplot(
  data = diamonds,
  mapping = aes(x = cut, color = clarity)
) +
  geom_bar(fill = NA, position = "identity")
```

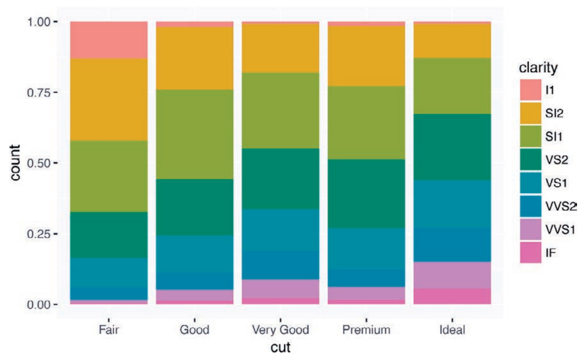




标识位置调整更适合 2D 几何对象，比如，点的标识位置调整是默认设置。

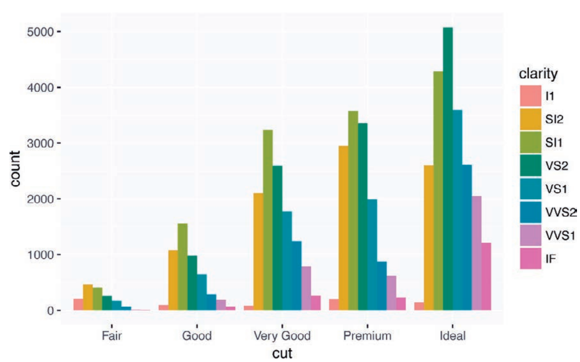
- `position = "fill"` 的效果与堆叠相似，但每组堆叠条形具有同样的高度，因此这种条形图可以非常轻松地比较各组间的比例：

```
ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = clarity),
    position = "fill"
  )
```

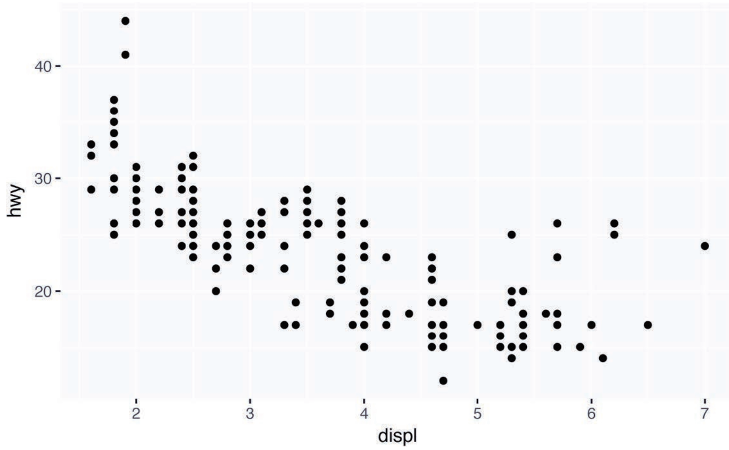


- `position = "dodge"` 将每组中的条形依次并列放置，这样可以非常轻松地比较每个条形表示的具体数值：

```
ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = clarity),
    position = "dodge"
  )
```



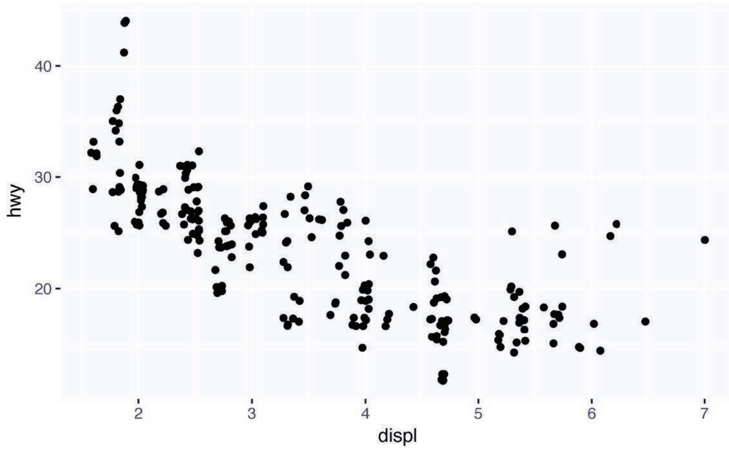
此外还有一种位置调整，虽然不适合条形图，但非常适合散点图。回忆一下我们的第一张散点图。你是否发现，虽然数据集中有 234 个观测值，但散点图中只显示了 126 个点？



因为 hwy 和 displ 的值都进行了舍入取整，所以这些点显示在一个网格上时，很多点彼此重叠了。这个问题称为**过绘制**。点的这种排列方式很难看出数据的聚集模式。数据点是均匀地分布在图中，还是存在 hwy 和 displ 的特殊组合，其中包括了 109 个点？

通过将位置调整方式设为“抖动”，可以避免这种网格化排列。position = "jitter" 为每个数据点添加一个很小的随机扰动，这样就可以将重叠的点分散开来，因为不可能有两个点会收到同样的随机扰动：

```
ggplot(data = mpg) +
  geom_point(
    mapping = aes(x = displ, y = hwy),
    position = "jitter"
  )
```



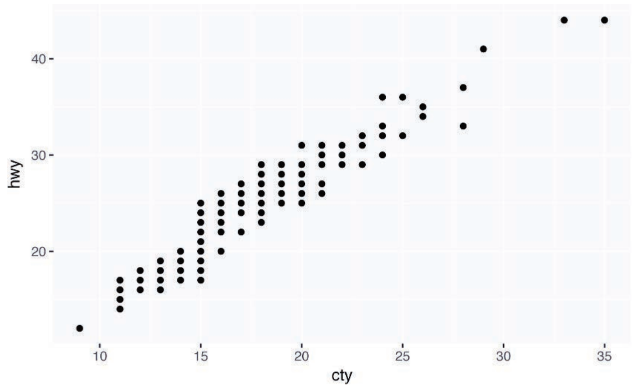
添加随机性来改善图形似乎是一种奇怪的方式，然而尽管这种方式会损失图形的精确性，但可以大大提高图形的启发性。因为这种操作的用处非常大，所以 ggplot2 提供了 `geom_point(position = "jitter")` 的一种快速实现方式：`geom_jitter()`。

要想了解有关位置调整的更多信息，可以查看每种调整方式的帮助页面：`?position_dodge`、`?position_fill`、`?position_identity`、`?position_jitter` 和 `?position_stack`。

练习

(1) 以下图形有什么问题？应该如何改善？

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point()
```



(2) `geom_jitter()` 使用哪些参数来控制抖动的程度？

(3) 对比 `geom_jitter()` 与 `geom_count()`。

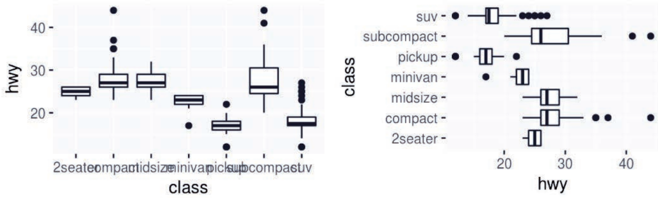
(4) `geom_boxplot()` 函数的默认位置调整方式是什么？创建 `mpg` 数据集的可视化表示来演示一下。

1.9 坐标系

坐标系可能是 ggplot2 中最复杂的部分。默认的坐标系是笛卡儿直角坐标系，可以通过其独立作用的 x 坐标和 y 坐标找到每个数据点。偶尔也会用到一些其他类型的坐标系。

- `coord_flip()` 函数可以交换 x 轴和 y 轴。当想要绘制水平箱线图时，这非常有用。它也非常适合使用长标签，但要想在 x 轴上不重叠地安排好它们是非常困难的：

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot()
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip()
```



- `coord_quickmap()` 函数可以为地图设置合适的纵横比。当使用 ggplot2 绘制空间数据时，这个函数特别重要（遗憾的是本书不涉及空间数据）：

```
nz <- map_data("nz")

ggplot(nz, aes(long, lat, group = group)) +
  geom_polygon(fill = "white", color = "black")

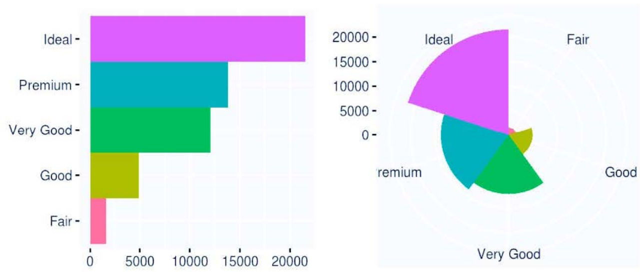
ggplot(nz, aes(long, lat, group = group)) +
  geom_polygon(fill = "white", color = "black") +
  coord_quickmap()
```



- `coord_polar()` 函数使用极坐标系。极坐标系可以揭示出条形图和鸡冠花图间的一种有趣联系：

```
bar <- ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = cut),
    show.legend = FALSE,
    width = 1
  ) +
  theme(aspect.ratio = 1) +
  labs(x = NULL, y = NULL)

bar + coord_flip()
bar + coord_polar()
```



练习

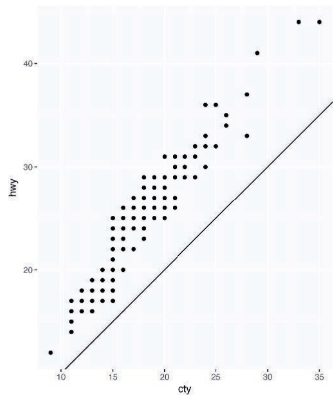
(1) 使用 `coord_polar()` 函数将堆叠式条形图转换为饼图。

(2) `labs()` 函数的功能是什么？阅读一下文档。

(3) `coord_quickmap()` 函数和 `coord_map()` 函数的区别是什么？

(4) 下图表明城市和公路燃油效率之间有什么关系？为什么 `coord_fixed()` 函数很重要？`geom_abline()` 函数的作用是什么？

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point() +
  geom_abline() +
  coord_fixed()
```



1.10 图形分层语法

在前面几节中，你学到的绝不仅仅是如何绘制散点图、条形图和箱线图，而是使用 ggplot2 绘制**任何**类型图形的基础知识。为了说明这一点，我们向前面的代码模板中添加位置调整、统计变换、坐标系和分面：

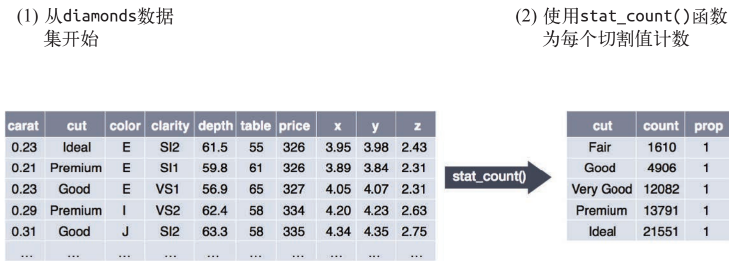
```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
     mapping = aes(<MAPPINGS>),
     stat = <STAT>,
     position = <POSITION>
  ) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

新模板有 7 个参数，即模板中尖括号内的部分。实际上，绘图时几乎不需要提供所有的 7 个参数，因为除了数据、映射和几何对象函数，ggplot2 为所有其他参数提供了非常有用的默认设置。

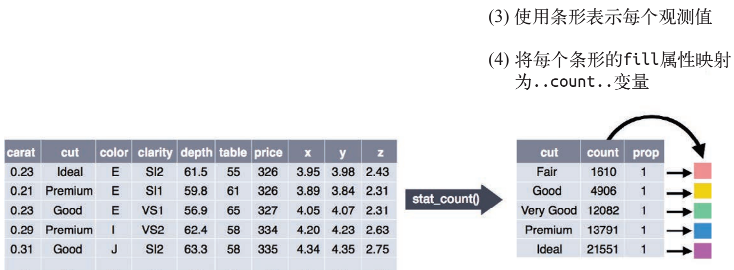
模板中的 7 个参数一同组成了图形语法，即用于建立图形的一个正式语法系统。你可以将任何图形精确地描述为数据集、几何对象、映射集合、统计变换、位置调整、坐标系和分

面模式的一个组合，图形语法正是基于这样的深刻理解构建出来的。

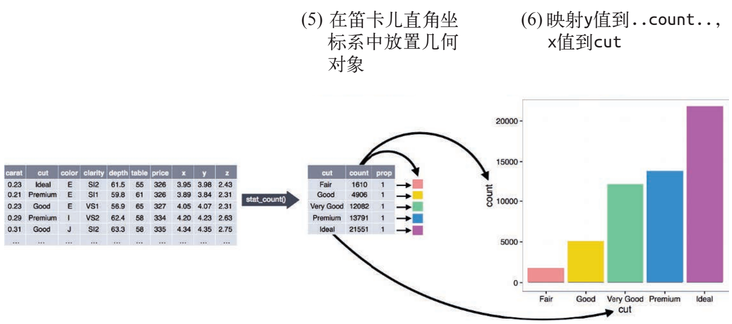
为了说明图形语法的工作方式，我们看一下如何从头开始构建一个基本图形：首先需要有一个数据集，然后（通过统计变换）将其转换为想要显示的信息。



接下来，你可以选择一个几何对象来表示转换后的数据中的每个观测值，然后选择几何对象的图形属性来表示数据中的变量，这会将每个变量的值映射为图形属性的水平。



下一步是选择放置几何对象的坐标系。你可以使用对象位置（对象本身的一个图形属性）来显示 x 变量和 y 变量的值。这样就生成了一张完整的图。但你还可以进一步调整几何对象在坐标系中的位置（位置调整），或者将图划分为多个子图（分面）。你还可以通过添加一个或多个附加图层对图进行扩展，其中每个附加图层都使用一个数据集、一个几何对象、一个映射集合、一个统计变换和一个位置调整。



你可以使用这种方法构建你能够想象到的**任何**图形。换句话说，你可以使用在本章中学到的代码模板来构建成千上万种独特的图形。

第 2 章

工作流：基础

现在你已经拥有了运行 R 代码的一些经验。我们没有介绍太多细节，但你肯定已经掌握了 R 的基础知识，否则你早已沮丧地将本书束之高阁了。当开始用 R 编程时，感到受挫是很自然的，因为 R 甚至对标点符号都非常严格，即使一个字符的错误也会导致问题。但是当有了一些心理准备后，你就可以心安理得地接受这些挫折，知道这是正常的，也是暂时的：每个人都会遇到困难，克服困难的唯一方法就是不断尝试。

在进一步学习之前，必须先确保你已经具有了运行 R 代码的坚实基础，并且掌握了 RStudio 中一些最有用的功能。

2.1 代码基础

为了让你尽快学会绘图，我们省略了一些基础知识，现在就来复习一下。你可以将 R 当作计算器来使用：

```
1 / 200 * 30
#> [1] 0.15
(59 + 73 + 2) / 3
#> [1] 44.7
sin(pi / 2)
#> [1] 1
```

你可以使用 `<-` 来创建新对象：

```
x <- 3 * 4
```

创建对象的所有 R 语句（即**赋值**语句）都有同样的形式：

```
object_name <- value
```

在阅读这行代码时，你可以在脑海中默念“某个对象名得到了某个值”。

你可能会进行大量的赋值操作，输入 <- 太痛苦了。但不要偷懒使用 =，虽然 = 确实也可以赋值，但之后会引起混淆。你可以使用 RStudio 快捷键：Alt+-（Alt 加上减号）。注意，RStudio 会自动在 <- 的两端加上空格，这是一个非常好的编码习惯。读代码是苦中作乐的一件事情，因此，用空格让你的眼睛稍感轻松吧。

2.2　对象名称

对象名称必须以字母开头，并且只能包含字母、数字、_ 和 .。如果想让对象名称具有描述性，那么就应该在使用多个单词时遵循某种命名惯例。我推荐使用 snake_case 命名法，也就是使用小写单词，并用 _ 分隔：

```
i_use_snake_case
otherPeopleUseCamelCase
some.people.use.periods
And_aFew.People_RENOUNCEconvention
```

我们将在第 14 章中继续讨论编码风格。

你可以通过输入对象名称来查看这个对象：

```
x
#> [1] 12
```

再进行赋值：

```
this_is_a_really_long_name <- 2.5
```

要想查看这个变量，可以使用 RStudio 的自动完成功能：输入“this”，按 Tab 键，继续输入字符直到完全匹配这个变量，然后按回车键。

哎呀，我们犯了一个错误！ this_is _a_really_long_name 的值应该是 3.5，而不是 2.5。这时可以使用另一种快捷键来修改对象。在命令窗口中输入“this”，然后按 Ctrl+ ↑ 。这样就可以列出所有输入过的以“this”开头的命令。使用箭头键上下移动，然后按回车键重新输入该命令。将 2.5 修改为 3.5，并按回车键。

再进行一次赋值：

```
r_rocks <- 2 ^ 3
```

查看一下这个对象：

```
r_rock
#> Error: object 'r_rock' not found
R_rocks
#> Error: object 'R_rocks' not found
```

R 和用户之间有一个隐含约定：R 可以替用户执行那些单调乏味的计算，但前提是用户必须输入完全精确的指令。不能有输入错误，还要区分大小写。

2.3　函数调用

R 中有大量内置函数，调用方式如下：

```
function_name(arg1 = val1, arg2 = val2, ...)
```

我们尝试使用 seq() 函数，它可以生成规则的数值序列，在学习这个函数的同时，我们还可以学习 RStudio 的更多有用功能。输入 se，并按 Tab 键。这时会弹出所有可能的自动完成命令。继续输入（“q”）以消除歧义，或者使用 ↑ 和 ↓ 箭头键来选择，以选定 seq() 函数。注意弹出的浮动提示信息，它可以告诉你这个函数的参数和作用。如果想要获得更多帮助，按 F1 键就可以在右下角窗格的帮助标签页中看到详细的帮助信息。

选定需要的函数后再按一次 Tab 键。RStudio 会为你自动添加开括号（(）和闭括号（)）。输入参数 1，10，然后按回车键：

```
seq(1, 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

输入以下代码，你会发现 RStudio 也会自动完成一对双引号以方便输入：

```
x <- "hello world"
```

引号和括号必须一直成对出现。RStudio 会尽力帮助我们，但还是有出错并导致不匹配的可能。如果出现不匹配，R 会显示一个 + 号：

```
> x <- "hello
+
```

+ 号表明 R 在等待继续输入；它认为你还没有完成输入。这通常意味着你漏掉了一个 " 或者)。你可以添加漏掉的部分，也可以按 Esc 键中止命令来重新输入。

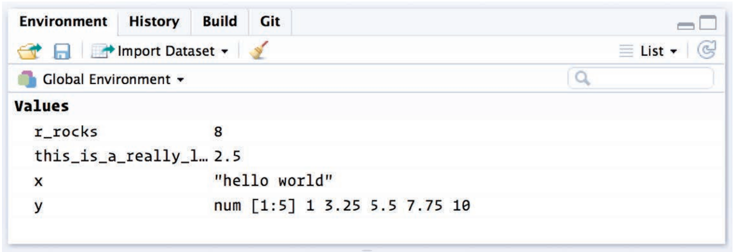
如果进行了一次赋值，R 不会显示出赋值结果。你最好立刻检查一下：

```
y <- seq(1, 10, length.out = 5)
y
#> [1]  1.00  3.25  5.50  7.75 10.00
```

这种常用的操作可以简化一下，用括号将赋值语句括起来就可以了，这样相当于连续执行赋值语句和“输出到屏幕”的操作：

```
(y <- seq(1, 10, length.out = 5))
#> [1]  1.00  3.25  5.50  7.75 10.00
```

现在看一下左上角窗格中的程序环境：



你可以在这里看到创建的所有对象。

练习

(1) 为什么以下代码不能正常运行？

```
my_variable <- 10
my_varıable
#> Error in eval(expr, envir, enclos):
#> object 'my_varıable' not found
```

仔细查看！（这个练习似乎没什么意义，但却是对思维的一种训练，它会让你意识到，在编程时，即使一点微小的区别也会导致程序无法正常运行。）

(2) 修改以下每段 R 代码，使其可以正常运行。

```
library(tidyverse)

ggplot(dota = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))

fliter(mpg, cyl = 8)
filter(diamond, carat > 3)
```

(3) 按 Alt+Shift+K 组合键会发生什么情况？如何使用菜单完成同样的操作？

第 3 章

使用dplyr进行数据转换

3.1 简介

可视化是生成见解的重要工具，但它需要数据格式完全符合我们的要求，这种情况是非常罕见的。一般来说，你需要创建一些新变量或者摘要统计量，还可能对变量进行重命名或者对观测值进行重新排序，以便数据更容易处理。你将在本章中学会如何进行这些甚至更多操作，本章将教会你如何使用 dplyr 包来转换数据，并介绍一个新的数据集：2013 年从纽约市出发的航班信息。

3.1.1 准备工作

本章将重点讨论如何使用 tidyverse 中的另一个核心 R 包——dplyr 包。我们使用 nycflights13 包中的数据来说明 dplyr 包的核心理念，并使用 ggplot2 来帮助我们理解数据。

```
library(nycflights13)
library(tidyverse)
```

加载 tidyverse 时，仔细查看输出的冲突信息，它会告诉你 dplyr 覆盖了基础 R 包中的哪些函数。如果想要在加载 dplyr 后使用这些函数的基础版本，那么你应该使用它们的完整名称：`stats::filter()` 和 `stats::lag()`。

3.1.2 nycflights13

为了介绍 dplyr 中的基本数据操作，我们需要使用 `nycflights13::flights`。这个数据框包含了 2013 年从纽约市出发的所有 336 776 次航班的信息。该数据来自于美国交通统计局，可以使用 `?flights` 查看其说明文档：

```
flights
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> 5  2013     1     1      554            600        -6
#> 6  2013     1     1      554            558        -4
#> # ... with 336,776 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#> #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

你或许会发现，这个数据框的输出和我们以前用过的其他数据框有一点差别：只显示了前几行和适合屏幕宽度的几列。（要想看到整个数据集，可以使用 `View(flights)` 在 RStudio 查看器中打开数据集。）输出有差别是因为 `flights` 是一个 tibble。tibble 也是一种数据框，只是进行了一些小小的修改，使其更适合在 tidyverse 中使用。现在，你不必关心它们之间的区别，本书的第二部分会更加详细地介绍 tibble。

你或许还会发现，列名下面有一行 3 个或 4 个字母的缩写。它们描述了每个变量的类型。

- `int` 表示整数型变量。
- `dbl` 表示双精度浮点数型变量，或称实数。
- `chr` 表示字符向量，或称字符串。
- `dttm` 表示日期时间（日期＋时间）型变量。

还有另外 3 种常用的变量类型，虽然没有在这个数据集中出现，但很快就会在本书后面遇到。

- `lgl` 表示逻辑型变量，是一个仅包括 `TRUE` 和 `FALSE` 的向量。
- `fctr` 表示因子，R 用其来表示具有固定数目的值的分类变量。
- `date` 表示日期型变量。

3.1.3　dplyr基础

我们将在本章中学习 5 个 dplyr 核心函数，它们可以帮助你解决数据处理中的绝大多数难题。

- 按值筛选观测（`filter()`）。
- 对行进行重新排序（`arrange()`）。
- 按名称选取变量（`select()`）。
- 使用现有变量的函数创建新变量（`mutate()`）。
- 将多个值总结为一个摘要统计量（`summarize()`）。

这些函数都可以和 `group_by()` 函数联合起来使用，`group_by()` 函数可以改变以上每个函数的作用范围，让其从在整个数据集上操作变为在每个分组上分别操作。这 6 个函数构成了

数据处理语言的基本操作。

前面 5 个函数的工作方式都是相同的。

(1) 第一个参数是一个数据框。

(2) 随后的参数使用变量名称（不带引号）描述了在数据框上进行的操作。

(3) 输出结果是一个新数据框。

利用以上这些属性可以很轻松地将多个简单步骤链接起来，从而得到非常复杂的结果。接下来我们将深入了解，看看如何使用这些操作。

3.2 使用filter()筛选行

`filter()` 函数可以基于观测的值筛选出一个观测子集。第一个参数是数据框名称，第二个参数以及随后的参数是用来筛选数据框的表达式。例如，我们可以使用以下代码筛选出 1 月 1 日的所有航班：

```
filter(flights, month == 1, day == 1)
#> # A tibble: 842 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> 5  2013     1     1      554            600        -6
#> 6  2013     1     1      554            558        -4
#> # ... with 836 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>,origin <chr>,
#> #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

如果运行这行代码，dplyr 就会执行筛选操作，并返回一个新数据框。dplyr 函数从来不修改输入，因此，如果想要保存函数结果，那么你就需要使用赋值操作符 `<-`：

```
jan1 <- filter(flights, month == 1, day == 1)
```

R 要么输出结果，要么将结果保存在一个变量中。如果想同时完成这两种操作，那么你可以用括号将赋值语句括起来：

```
(dec25 <- filter(flights, month == 12, day == 25))
#> # A tibble: 719 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013    12    25      456            500        -4
#> 2  2013    12    25      524            515         9
#> 3  2013    12    25      542            540         2
#> 4  2013    12    25      546            550        -4
#> 5  2013    12    25      556            600        -4
#> 6  2013    12    25      557            600        -3
```

```
#> # ... with 713 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>,origin <chr>,
#> #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

3.2.1 比较运算符

为了有效地进行筛选，你必须知道如何使用比较运算符来选择观测。R 提供了一套标准的比较运算符：`>`、`>=`、`<`、`<=`、`!=`（不等于）和 `==`（等于）。

当开始使用 R 时，最容易犯的错误就是使用 `=` 而不是 `==` 来测试是否相等。当出现这种情况时，你会收到一条有启发性的错误消息：

```
filter(flights, month = 1)
#> Error: filter() takes unnamed arguments. Do you need `==`?
```

在使用 `==` 进行比较时，你可能还会遇到另一个常见问题：浮点数。下面的结果可能会令你目瞪口呆：

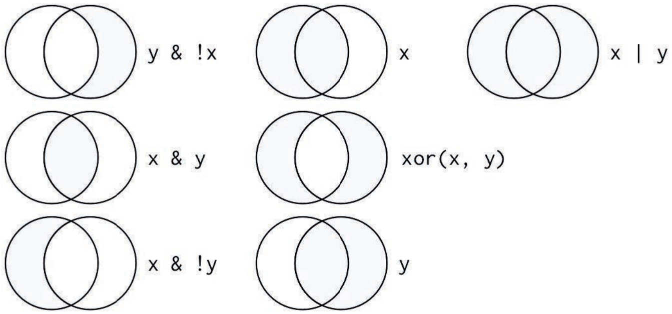
```
sqrt(2) ^ 2 == 2
#> [1] FALSE
1/49 * 49 == 1
#> [1] FALSE
```

计算机使用的是有限精度运算（显然无法存储无限位的数），因此请记住，你看到的每个数都是一个近似值。比较浮点数是否相等时，不能使用 `==`，而应该使用 `near()`：

```
near(sqrt(2) ^ 2, 2)
#> [1] TRUE
near(1 / 49 * 49, 1)
#> [1] TRUE
```

3.2.2 逻辑运算符

`filter()` 中的多个参数是由“与”组合起来的：每个表达式都必须为真才能让一行观测包含在输出中。如果要实现其他类型的组合，你需要使用布尔运算符：`&` 表示“与”、`|` 表示“或”、`!` 表示“非”。下图给出了布尔运算的完整集合。



以下代码可以找出 11 月或 12 月出发的所有航班：

```
filter(flights, month == 11 | month == 12)
```

表达式中的运算顺序和语言中的是不一样的。你不能写成 `filter(flights, month == 11 | 12)` 这种形式。这种形式的文字翻译确实是"找出 11 月或 12 月出发的所有航班"，但在代码中则不是这个意思，代码中的含义是找出所有出发月份为 `11 | 12` 的航班。`11 | 12` 这个逻辑表达式的值为 `TRUE`，在数字语境中（如本例），`TRUE` 就是 `1`，所以这段代码找出的不是 11 月或 12 月出发的航班，而是 1 月出发的所有航班。真是够绕的！

这种问题有一个有用的简写形式：`x %in% y`。这会选取出 x 是 y 中的一个值时的所有行。我们可以使用这种形式重写上面的代码：

```
nov_dec <- filter(flights, month %in% c(11, 12))
```

有时你可以使用德摩根定律将复杂的筛选条件进行简化：`!(x & y)` 等价于 `!x | !y`、`!(x | y)` 等价于 `!x & !y`。例如，如果想要找出延误时间（到达或出发）不多于 2 小时的航班，那么使用以下两种筛选方式均可：

```
filter(flights, !(arr_delay > 120 | dep_delay > 120))
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

除 `&` 和 `|` 之外，R 中还有 `&&` 和 `||` 运算符。先不要使用这两个运算符！ 14.4 节会说明何时使用它们。

只要 `filter()` 函数中使用的是复杂的、包含多个部分的表达式，就需要考虑用一个明确的变量来代替它。这样检查代码会容易很多。我们很快就会介绍如何创建新变量。

3.2.3　缺失值

R 的一个重要特征使得比较运算更加复杂，这个特征就是缺失值，或称 `NA`（not available，不可用）。`NA` 表示未知的值，因此缺失值是"可传染的"。如果运算中包含了未知值，那么运算结果一般来说也是个未知值：

```
NA > 5
#> [1] NA
10 == NA
#> [1] NA
NA + 10
#> [1] NA
NA / 2
#> [1] NA
```

最令人费解的是以下这个结果：

```
NA == NA
#> [1] NA
```

要想理解为什么会这样，最容易的方式是加入一点背景知识：

```
# 令x为Mary的年龄。我们不知道她有多大。
x <- NA
```

```
# 令y为John的年龄。我们不知道他有多大。
y <- NA

# John和Mary的年龄是相同的吗？
x == y
#> [1] NA
# 我们不知道！
```

如果想要确定一个值是否为缺失值，可以使用 `is.na()` 函数：

```
is.na(x)
#> [1] TRUE
```

`filter()` 只能筛选出条件为 `TRUE` 的行；它会排除那些条件为 `FALSE` 和 `NA` 的行。如果想保留缺失值，可以明确指出：

```
df <- tibble(x = c(1, NA, 3))
filter(df, x > 1)
#> # A tibble: 1 × 1
#>       x
#>   <dbl>
#> 1     3
filter(df, is.na(x) | x > 1)
#> # A tibble: 2 × 1
#>       x
#>   <dbl>
#> 1    NA
#> 2     3
```

3.2.4 练习

(1) 找出满足以下条件的所有航班。

a. 到达时间延误 2 小时或更多的航班。
b. 飞往休斯顿（IAH 机场或 HOU 机场）的航班。
c. 由联合航空（United）、美利坚航空（American）或三角洲航空（Delta）运营的航班。
d. 夏季（7 月、8 月和 9 月）出发的航班。
e. 到达时间延误超过 2 小时，但出发时间没有延误的航班。
f. 延误至少 1 小时，但飞行过程弥补回 30 分钟的航班。
g. 出发时间在午夜和早上 6 点之间（包括 0 点和 6 点）的航班。

(2) dplyr 中对筛选有帮助的另一个函数是 `between()`。它的作用是什么？你能使用这个函数来简化解决前面问题的代码吗？

(3) `dep_time` 有缺失值的航班有多少？其他变量的缺失值情况如何？这样的行表示什么情况？

(4) 为什么 `NA ^ 0` 的值不是 `NA`？为什么 `NA | TRUE` 的值不是 `NA`？为什么 `FALSE & NA` 的值不是 `NA`？你能找出一般规律吗？（`NA * 0` 则是精妙的反例！）

3.3 使用arrange()排列行

arrange() 函数的工作方式与 filter() 函数非常相似，但前者不是选择行，而是改变行的顺序。它接受一个数据框和一组作为排序依据的列名（或者更复杂的表达式）作为参数。如果列名不只一个，那么就使用后面的列在前面排序的基础上继续排序：

```
arrange(flights, year, month, day)
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> 5  2013     1     1      554            600        -6
#> 6  2013     1     1      554            558        -4
#> # ... with 3.368e+05 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#> #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

使用 desc() 可以按列进行降序排序：

```
arrange(flights, desc(arr_delay))
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     9      641            900      1301
#> 2  2013     6    15     1432           1935      1137
#> 3  2013     1    10     1121           1635      1126
#> 4  2013     9    20     1139           1845      1014
#> 5  2013     7    22      845           1600      1005
#> 6  2013     4    10     1100           1900       960
#> # ... with 3.368e+05 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#> #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>,
```

缺失值总是排在最后：

```
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
#> # A tibble: 3 × 1
#>       x
#>   <dbl>
#> 1     2
#> 2     5
#> 3    NA
arrange(df, desc(x))
#> # A tibble: 3 × 1
#>       x
#>   <dbl>
```

```
#> 1      5
#> 2      2
#> 3     NA
```

练习

(1) 如何使用 `arrange()` 将缺失值排在最前面？（提示：使用 `is.na()`。）

(2) 对 `flights` 排序以找出延误时间最长的航班。找出出发时间最早的航班。

(3) 对 `flights` 排序以找出速度最快的航班。

(4) 哪个航班的飞行时间最长？哪个最短？

3.4 使用select()选择列

如今，数据集有几百甚至几千个变量已经司空见惯。这种情况下，如何找出真正感兴趣的那些变量经常是我们面临的第一个挑战。通过基于变量名的操作，`select()` 函数可以让你快速生成一个有用的变量子集。

`select()` 函数对于航班数据不是特别有用，因为其中只有 19 个变量，但你还是可以通过这个数据集了解一下 `select()` 函数的大致用法：

```
# 按名称选择列
select(flights, year, month, day)
#> # A tibble: 336,776 × 3
#>     year month   day
#>    <int> <int> <int>
#> 1  2013     1     1
#> 2  2013     1     1
#> 3  2013     1     1
#> 4  2013     1     1
#> 5  2013     1     1
#> 6  2013     1     1
#> # ... with 3.368e+05 more rows

# 选择“year”和“day”之间的所有列（包括“year”和“day”）
select(flights, year:day)
#> # A tibble: 336,776 × 3
#>     year month   day
#>    <int> <int> <int>
#> 1  2013     1     1
#> 2  2013     1     1
#> 3  2013     1     1
#> 4  2013     1     1
#> 5  2013     1     1
#> 6  2013     1     1
#> # ... with 3.368e+05 more rows

# 选择不在“year”和“day”之间的所有列（不包括“year”和“day”）
select(flights, -(year:day))
#> # A tibble: 336,776 × 16
```

```
#>   dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>      <int>          <int>     <dbl>    <int>          <int>
#> 1      517            515         2      830            819
#> 2      533            529         4      850            830
#> 3      542            540         2      923            850
#> 4      544            545        -1     1004           1022
#> 5      554            600        -6      812            837
#> 6      554            558        -4      740            728
#> # ... with 3.368e+05 more rows, and 12 more variables:
#> #   arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>,  hour <dbl>, minute <dbl>,
#> #   time_hour <dttm>
```

还可以在 select () 函数中使用一些辅助函数。

- starts_with("abc")：匹配以"abc"开头的名称。
- ends_with("xyz")：匹配以"xyz"结尾的名称。
- contains("ijk")：匹配包含"ijk"的名称。
- matches("(.)\\1")：选择匹配正则表达式的那些变量。这个正则表达式会匹配名称中有重复字符的变量。你将在第 10 章中学习到更多关于正则表达式的知识。
- num_range("x", 1:3)：匹配 x1、x2 和 x3。

使用 ?select 命令可以获取更多信息。

select() 可以重命名变量，但我们很少这样使用它，因为这样会丢掉所有未明确提及的变量。我们应该使用 select() 函数的变体 rename() 函数来重命名变量，以保留所有未明确提及的变量：

```
rename(flights, tail_num = tailnum)
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> 5  2013     1     1      554            600        -6
#> 6  2013     1     1      554            558        -4
#> # ... with 3.368e+05 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tail_num <chr>,
#> #   origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>,
#> #   time_hour <dttm>
```

另一种用法是将 select() 函数和 everything() 辅助函数结合起来使用。当想要将几个变量移到数据框开头时，这种用法非常奏效：

```
select(flights, time_hour, air_time, everything())
#> # A tibble: 336,776 × 19
#>             time_hour air_time year month    day dep_time
#>                <dttm>    <dbl> <int> <int> <int>    <int>
```

```
#> 1 2013-01-01 05:00:00      227  2013     1     1      517
#> 2 2013-01-01 05:00:00      227  2013     1     1      533
#> 3 2013-01-01 05:00:00      160  2013     1     1      542
#> 4 2013-01-01 05:00:00      183  2013     1     1      544
#> 5 2013-01-01 06:00:00      116  2013     1     1      554
#> 6 2013-01-01 05:00:00      150  2013     1     1      554
#> # ... with 3.368e+05 more rows, and 13 more variables:
#> #   sched_dep_time <int>, dep_delay <dbl>, arr_time <int>,
#> #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#> #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#> #   distance <dbl>, hour <dbl>, minute <dbl>
```

练习

(1) 从 flights 数据集中选择 dep_time、dep_delay、arr_time 和 arr_delay，通过头脑风暴找出尽可能多的方法。

(2) 如果在 select() 函数中多次计入一个变量名，那么会发生什么情况？

(3) one_of() 函数的作用是什么？为什么它结合以下向量使用时非常有用？

```
vars <- c(
  "year", "month", "day", "dep_delay", "arr_delay"
)
```

(4) 以下代码的运行结果是否出乎意料？选择辅助函数处理大小写的默认方式是什么？如何改变默认方式？

```
select(flights, contains("TIME"))
```

3.5 使用mutate()添加新变量

除了选择现有的列，我们还经常需要添加新列，新列是现有列的函数。这就是 mutate() 函数的作用。

mutate() 总是将新列添加在数据集的最后，因此我们需要先创建一个更狭窄的数据集，以便能够看到新变量。记住，当使用 RStudio 时，查看所有列的最简单的方法就是使用 View() 函数：

```
flights_sml <- select(flights,
  year:day,
  ends_with("delay"),
  distance,
  air_time
)
mutate(flights_sml,
  gain = arr_delay - dep_delay,
  speed = distance / air_time * 60
)
#> # A tibble: 336,776 × 9
#>    year month   day dep_delay arr_delay distance air_time
#>   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
```

```
#> 1  2013     1     1         2        11     1400      227
#> 2  2013     1     1         4        20     1416      227
#> 3  2013     1     1         2        33     1089      160
#> 4  2013     1     1        -1       -18     1576      183
#> 5  2013     1     1        -6       -25      762      116
#> 6  2013     1     1        -4        12      719      150
#> # ... with 3.368e+05 more rows, and 2 more variables:
#> #  gain <dbl>, speed <dbl>
```

一旦创建，新列就可以立即使用：

```
mutate(flights_sml,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
#> # A tibble: 336,776 × 10
#>    year month   day dep_delay arr_delay distance air_time
#>   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
#> 1  2013     1     1         2        11     1400      227
#> 2  2013     1     1         4        20     1416      227
#> 3  2013     1     1         2        33     1089      160
#> 4  2013     1     1        -1       -18     1576      183
#> 5  2013     1     1        -6       -25      762      116
#> 6  2013     1     1        -4        12      719      150
#> # ... with 3.368e+05 more rows, and 3 more variables:
#> #  gain <dbl>, hours <dbl>, gain_per_hour <dbl>
```

如果只想保留新变量，可以使用 `transmute()` 函数：

```
transmute(flights,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
#> # A tibble: 336,776 × 3
#>    gain hours gain_per_hour
#>   <dbl> <dbl>         <dbl>
#> 1     9  3.78          2.38
#> 2    16  3.78          4.23
#> 3    31  2.67         11.62
#> 4   -17  3.05         -5.57
#> 5   -19  1.93         -9.83
#> 6    16  2.50          6.40
#> # ... with 3.368e+05 more rows
```

3.5.1　常用创建函数

创建新变量的多种函数可供你同 `mutate()` 一同使用。最重要的一点是，这种函数必须是向量化的：它必须接受一个向量作为输入，并返回一个向量作为输出，而且输入向量与输出向量具有同样数目的分量。我们无法列出所有可能用到的创建函数，但可以介绍一下那些比较常用的。

算术运算符：+、-、*、/、^

它们都是向量化的，使用所谓的“循环法则”。如果一个参数比另一个参数短，那么前者会自动扩展到同样的长度。当某个参数是单个数值时，这种方式是最有效的：air_time / 60、hours * 60 + minute 等。

算术运算符的另一用途是与我们后面将很快学到的聚集函数结合起来使用。例如，x / sum(x) 可以计算出各个分量在总数中的比例，y - mean(y) 可以计算出分量与均值之间的差值。

模运算符：%/% 和 %%

%/%（整数除法）和 %%（求余）满足 x == y * (x %/% y) + (x %% y)。模运算非常好用，因为它可以拆分整数。例如，在航班数据集中，你可以根据 dep_time 计算出 hour 和 minute：

```
transmute(flights,
  dep_time,
  hour = dep_time %/% 100,
  minute = dep_time %% 100
)
#> # A tibble: 336,776 × 3
#>   dep_time  hour minute
#>      <int> <dbl>  <dbl>
#> 1      517     5     17
#> 2      533     5     33
#> 3      542     5     42
#> 4      544     5     44
#> 5      554     5     54
#> 6      554     5     54
#> # ... with 3.368e+05 more rows
```

对数函数：log()、log2() 和 log10()

在处理取值范围横跨多个数量级的数据时，对数是特别有用的一种转换方式。它还可以将乘法转换成加法，我们将在本书的第四部分中介绍这个功能。

其他条件相同的情况下，我推荐使用 log2() 函数，因为很容易对其进行解释：对数标度的数值增加 1 个单位，意味着初始数值加倍；减少 1 个单位，则意味着初始数值减半。

偏移函数

lead() 和 lag() 函数可以返回一个序列的领先值和滞后值。它们可以计算出序列的移动差值（如 x - lag(x)）或发现序列何时发生了变化（x != lag(x)）。它们与 group_by() 组合使用时特别有用，你很快就会学到 group_by() 这个函数：

```
(x <- 1:10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
lag(x)
#>  [1] NA  1  2  3  4  5  6  7  8  9
lead(x)
#>  [1]  2  3  4  5  6  7  8  9 10 NA
```

累加和滚动聚合

R 提供了计算累加和、累加积、累加最小值和累加最大值的函数：cumsum()、cumprod()、cummin() 和 cummax()；dplyr 还提供了 cummean() 函数以计算累加均值。如果想要计算滚动聚合（即滚动窗口求和），那么可以尝试使用 RcppRoll 包：

```
x
#>  [1]  1  2  3  4  5  6  7  8  9 10
cumsum(x)
#>  [1]  1  3  6 10 15 21 28 36 45 55
cummean(x)
#>  [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

逻辑比较：<、<=、>、>= 和 !=

如果需要进行一系列复杂的逻辑运算，那么最好将中间结果保存在新变量中，这样就可以检查是否每一步都符合预期。

排秩

排秩函数有很多，但你应该从 min_rank() 函数开始，它可以完成最常用的排秩任务（如第一、第二、第三、第四）。默认的排秩方式是，最小的值获得最前面的名次，使用 desc(x) 可以让最大的值获得最前面的名次：

```
y <- c(1, 2, 2, NA, 3, 4)
min_rank(y)
#> [1]  1  2  2 NA  4  5
min_rank(desc(y))
#> [1]  5  3  3 NA  2  1
```

如果 min_rank() 无法满足需要，那么可以看一下其变体 row_number()、dense_rank()、percent_rank()、cume_dist() 和 ntile()。可以查看它们的帮助页面以获得更多信息：

```
row_number(y)
#> [1]  1  2  3 NA  4  5
dense_rank(y)
#> [1]  1  2  2 NA  3  4
percent_rank(y)
#> [1] 0.00 0.25 0.25   NA 0.75 1.00
cume_dist(y)
#> [1] 0.2 0.6 0.6  NA 0.8 1.0
```

3.5.2 练习

(1) 虽然现在的 dep_time 和 sched_dep_time 变量方便阅读，但不适合计算，因为它们实际上并不是连续型数值。将它们转换成一种更方便的表示形式，即从午夜开始的分钟数。

(2) 比较 air_time 和 arr_time - dep_time。你期望看到什么？实际又看到了什么？如何解决这个问题？

(3) 比较 dep_time、sched_dep_time 和 dep_delay。你期望这 3 个数值之间具有何种关系？

(4) 使用排秩函数找出 10 个延误时间最长的航班。如何处理名次相同的情况？仔细阅读 min_rank() 的帮助文件。

(5) 1:3 + 1:10 会返回什么？为什么？

(6) R 提供了哪些三角函数？

3.6 使用summarize()进行分组摘要

最后一个核心函数是 summarize()，它可以将数据框折叠成一行：

```
summarize(flights, delay = mean(dep_delay, na.rm = TRUE))
#> # A tibble: 1 × 1
#>   delay
#>   <dbl>
#> 1  12.6
```

（我们很快就会解释 na.rm = TRUE 的含义。）

如果不与 group_by() 一起使用，那么 summarize() 也就没什么大用。group_by() 可以将分析单位从整个数据集更改为单个分组。接下来，在分组后的数据框上使用 dplyr 函数时，它们会自动地应用到每个分组。例如，如果对按日期分组的一个数据框应用与上面完全相同的代码，那么我们就可以得到每日平均延误时间：

```
by_day <- group_by(flights, year, month, day)
summarize(by_day, delay = mean(dep_delay, na.rm = TRUE))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>     year month   day delay
#>    <int> <int> <int> <dbl>
#> 1  2013     1     1 11.55
#> 2  2013     1     2 13.86
#> 3  2013     1     3 10.99
#> 4  2013     1     4  8.95
#> 5  2013     1     5  5.73
#> 6  2013     1     6  7.15
#> # ... with 359 more rows
```

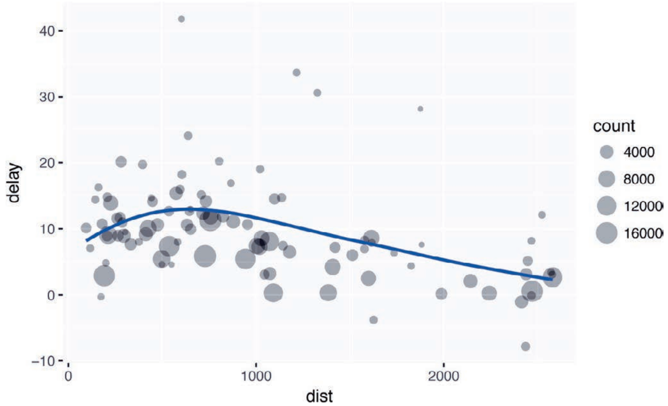
group_by() 和 summarize() 的组合构成了使用 dplyr 包时最常用的操作之一：分组摘要。在对其进行更深入的讨论之前，我们需要先介绍一个功能强大的新概念：管道。

3.6.1 使用管道组合多种操作

假设我们想要研究每个目的地的距离和平均延误时间之间的关系。使用已经了解的 dplyr 包功能，你可能会写出以下代码：

```
by_dest <- group_by(flights, dest)
delay <- summarize(by_dest,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE)
)
delay <- filter(delay, count > 20, dest != "HNL")
```

```
# 750英里内，平均延误时间会随着距离的增加而增加，接着会随着距离的增加而减少。随着飞
  行距离的增加，延误时间有可能会在飞行中弥补回来吗？
ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess'
```



完成数据准备需要 3 步。

(1) 按照目的地对航班进行分组。

(2) 进行摘要统计，计算距离、平均延误时间和航班数量。

(3) 通过筛选除去噪声点和火奴鲁鲁机场，因为到达该机场的距离几乎是到离它最近机场的距离的 2 倍。

这段代码写起来有点令人泄气，因为不得不对每个中间数据框命名，尽管我们根本不关心这一点。命名是很难的，这样做会影响我们的分析速度。

解决这个问题的另一种方法是使用管道，`%>%`：

```
delays <- flights %>%
  group_by(dest) %>%
  summarize(
    count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE)
) %>%
filter(count > 20, dest != "HNL")
```

这种方法的重点在于转换的过程，而不是转换的对象，这使得代码具有更好的可读性。你可以将其读作一串命令式语句：分组，然后摘要统计，然后进行筛选。在阅读代码时，`%>%` 最好读作“然后”。

使用这种方法时，`x %>% f(y)` 会转换为 `f(x, y)`，`x %>% f(y) %>% g(z)` 会转换为 `g(f(x, y), z)`，以此类推。你可以使用管道重写多种操作，将其变为能够从左到右或从上到下阅读。从现在开始，我们会频繁使用管道方式，因为它可以显著提高代码的可读性，我们将在第 13 章中对其进行更详细的介绍。

支持管道操作是 tidyverse 中的 R 包的核心原则之一。唯一的例外就是 ggplot2：它是在发现管道方式前开发的。ggplot2 的下一个版本 ggvis 支持管道操作，遗憾的是其还没有达到成熟完备的程度。

3.6.2 缺失值

我们在前面使用了参数 `na.rm`，你应该非常想要知道其含义。如果没有设置这个参数，会发生什么情况呢？

```
flights %>%
  group_by(year, month, day) %>%
  summarize(mean = mean(dep_delay))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day  mean
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1    NA
#> 2  2013     1     2    NA
#> 3  2013     1     3    NA
#> 4  2013     1     4    NA
#> 5  2013     1     5    NA
#> 6  2013     1     6    NA
#> # ... with 359 more rows
```

我们会得到很多缺失值！这是因为聚合函数遵循缺失值的一般规则：如果输入中有缺失值，那么输出也会是缺失值。好在所有聚合函数都有一个 `na.rm` 参数，它可以在计算前除去缺失值：

```
flights %>%
  group_by(year, month, day) %>%
  summarize(mean = mean(dep_delay, na.rm = TRUE))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day  mean
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1 11.55
#> 2  2013     1     2 13.86
#> 3  2013     1     3 10.99
#> 4  2013     1     4  8.95
#> 5  2013     1     5  5.73
#> 6  2013     1     6  7.15
#> # ... with 359 more rows
```

在这个示例中，缺失值表示取消的航班，我们也可以通过先去除取消的航班来解决缺失值问题。保存这个数据集，以便我们可以在接下来的几个示例中继续使用：

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled %>%
```

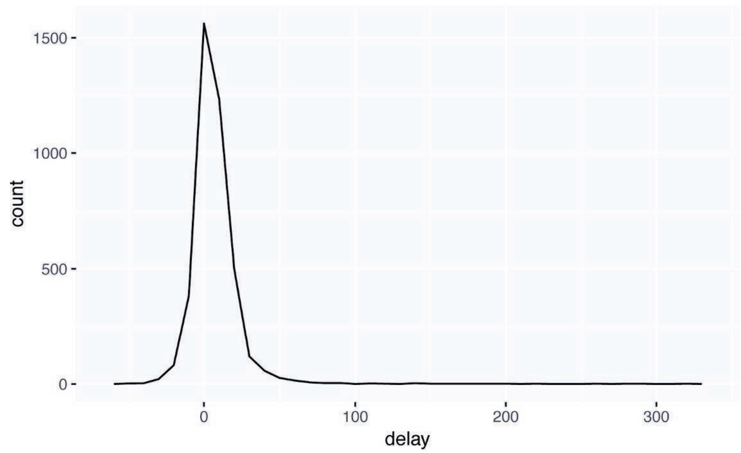
```
  group_by(year, month, day) %>%
  summarize(mean = mean(dep_delay))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day  mean
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1 11.44
#> 2  2013     1     2 13.68
#> 3  2013     1     3 10.91
#> 4  2013     1     4  8.97
#> 5  2013     1     5  5.73
#> 6  2013     1     6  7.15
#> # ... with 359 more rows
```

3.6.3 计数

聚合操作中包括一个计数（`n()`）或非缺失值的计数（`sum(!is_na())`）是个好主意。这样你就可以检查一下，以确保自己没有基于非常少量的数据作出结论。例如，我们查看一下具有最长平均延误时间的飞机（通过机尾编号进行识别）：

```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarize(
    delay = mean(arr_delay)
  )

ggplot(data = delays, mapping = aes(x = delay)) +
  geom_freqpoly(binwidth = 10)
```

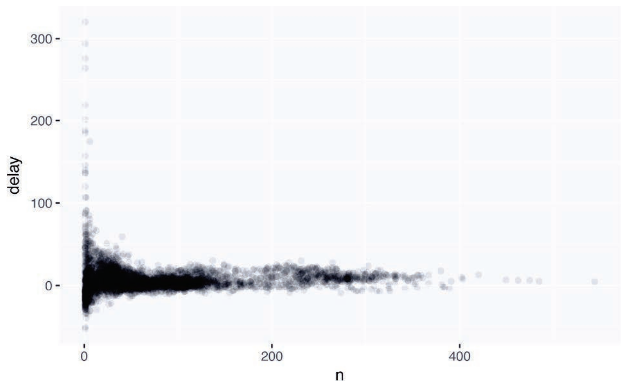


哇！有些飞机的**平均**延误时间长达 5 小时（300 分钟）！

这个情况确实有些微妙。我们可以画一张航班数量和平均延误时间的散点图，以便获得更深刻的理解：

```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarize(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )

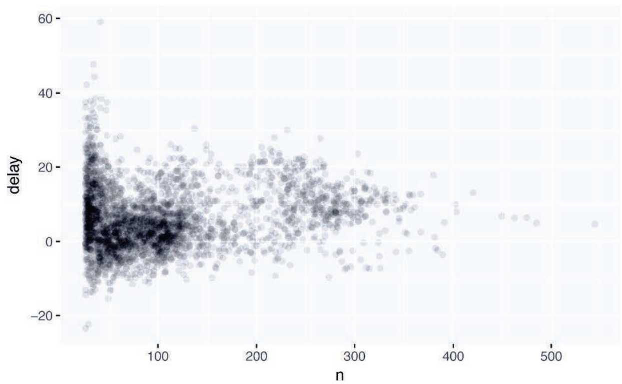
ggplot(data = delays, mapping = aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```



结果并不出乎意料，当航班数量非常少时，平均延误时间的变动特别大。这张图的形状非常能够说明问题：当绘制均值（或其他摘要统计量）和分组规模的关系时，你总能看到随着样本量的增加，变动在不断减小。

查看此类图形时，通常应该筛选掉那些观测数量非常少的分组，这样你就可以避免受到特别小的分组中的极端变动的影响，进而更好地发现数据模式。这就是以下代码要做的工作，同时还展示了将 ggplot2 集成到 dplyr 工作流中的一种有效方式。从 %>% 过渡到 + 会令人感到不适应，但掌握其中的要领后，这种写法是非常方便的：

```
delays %>%
  filter(n > 25) %>%
  ggplot(mapping = aes(x = n, y = delay)) +
    geom_point(alpha = 1/10)
```





RStudio 技巧：Ctrl+Shift+P 是非常有用的组合键，可以将上一次从编辑器发送到控制台的代码段重新发送一次。例如，当在上个示例中试验 n 的值时，使用这个组合键就特别方便。你可以使用 Ctrl+Enter 将整段代码发送到控制台，然后修改 n 的值，再按 Ctrl+Shift+P 就可以重新发送整段代码。

这种数据模式还有另外一种常见的变体。我们看一下棒球击球手的平均表现与击球次数之间的关系。我们使用 Lahman 包中的数据来计算大联盟的每个棒球队员的打击率（安打数 / 打数）。

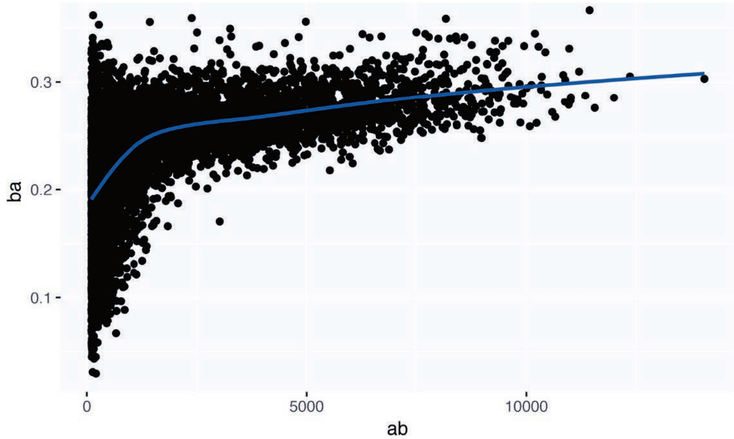
当我绘制出击球手的能力（用打击率 ba 衡量）与击球机会数量（用打数 ab 衡量）之间的关系时，你可以看到两种模式。

- 同上，数据点越多，聚合值的变动就越小。
- 能力（ba）和击球机会数量（ab）之间存在正相关。这是因为球队会控制击球手的出场，很显然，球队会优先选择最好的队员。

```
# 转换成tibble，以便输出更美观
batting <- as_tibble(Lahman::Batting)

batters <- batting %>%
  group_by(playerID) %>%
  summarize(
    ba = sum(H, na.rm = TRUE) / sum(AB, na.rm = TRUE),
    ab = sum(AB, na.rm = TRUE)
  )

batters %>%
  filter(ab > 100) %>%
  ggplot(mapping = aes(x = ab, y = ba)) +
    geom_point() +
    geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'gam'
```



这对球员排名也有重要影响。如果只是使用 desc(ba) 进行排序，明显受益的将是具有最好打击率的球员，而不是能力最高的球员：

```
batters %>%
  arrange(desc(ba))
#> # A tibble: 18,659 × 3
#>    playerID    ba    ab
#>       <chr> <dbl> <int>
#> 1 abramge01     1     1
#> 2 banisje01     1     1
#> 3 bartocl01     1     1
#> 4  bassdo01     1     1
#> 5 birasst01     1     2
#> 6 bruneju01     1     1
#> # ... with 1.865e+04 more rows
```

你可以在 http://varianceexplained.org/r/empirical_bayes_baseball/ 和 http://www.evanmiller.org/how-not-to-sort-by-average-rating.html 中找到有关这个问题的精彩解释。

3.6.4 常用的摘要函数

只使用均值、计数和求和是远远不够的，R 中还提供了很多其他的常用的摘要函数。

位置度量

我们已经使用过 mean(x)，但 median(x) 也非常有用。均值是总数除以个数；中位数则是这样一个值：50% 的 x 大于它，同时 50% 的 x 小于它。

有时候需要将聚合函数和逻辑筛选组合起来使用。我们还没有讨论过这种取数据子集的方法，15.5.2 节将对此进行深入介绍。

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(
    # 平均延误时间：
    avg_delay1 = mean(arr_delay),
    # 平均正延误时间：
    avg_delay2 = mean(arr_delay[arr_delay > 0])
  )
#> Source: local data frame [365 x 5]
#> Groups: year, month [?]
#>
#>    year month   day avg_delay1 avg_delay2
#>   <int> <int> <int>      <dbl>      <dbl>
#> 1  2013     1     1      12.65       32.5
#> 2  2013     1     2      12.69       32.0
#> 3  2013     1     3       5.73       27.7
#> 4  2013     1     4      -1.93       28.3
#> 5  2013     1     5      -1.53       22.6
#> 6  2013     1     6       4.24       24.4
#> # ... with 359 more rows
```

分散程度度量：sd(x)、IQR(x) 和 mad(x)

均方误差（又称标准误差，standard deviation，sd）是分散程度的标准度量方式。四分位距 IQR() 和绝对中位差 mad(x) 基本等价，更适合有离群点的情况：

```
# 为什么到某些目的地的距离比到其他目的地更多变？
not_cancelled %>%
  group_by(dest) %>%
  summarize(distance_sd = sd(distance)) %>%
  arrange(desc(distance_sd))
#> # A tibble: 104 × 2
#>     dest distance_sd
#>    <chr>       <dbl>
#> 1   EGE        10.54
#> 2   SAN        10.35
#> 3   SFO        10.22
#> 4   HNL        10.00
#> 5   SEA         9.98
#> 6   LAS         9.91
#> # ... with 98 more rows
```

秩的度量：min(x)、quantile(x, 0.25) 和 max(x)

分位数是中位数的扩展。例如，quantile(x, 0.25) 会找出 x 中按从小到大顺序大于前 25% 而小于后 75% 的值：

```
# 每天最早和最晚的航班何时出发？
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(
    first = min(dep_time),
    last = max(dep_time)
  )
#> Source: local data frame [365 x 5]
#> Groups: year, month [?]
#>
#>     year month   day first  last
#>    <int> <int> <int> <int> <int>
#> 1  2013     1     1   517  2356
#> 2  2013     1     2    42  2354
#> 3  2013     1     3    32  2349
#> 4  2013     1     4    25  2358
#> 5  2013     1     5    14  2357
#> 6  2013     1     6    16  2355
#> # ... with 359 more rows
```

定位度量：first(x)、nth(x, 2) 和 last(x)

这几个函数的作用与 x[1]、x[2] 和 x[length(x)] 相同，只是当定位不存在时（比如尝试从只有两个元素的分组中得到第三个元素），前者允许你设置一个默认值。例如，我们可以找出每天最早和最晚出发的航班：

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(
    first_dep = first(dep_time),
```

```
    last_dep = last(dep_time)
  )
#> Source: local data frame [365 x 5]
#> Groups: year, month [?]
#>
#>    year month   day first_dep last_dep
#>   <int> <int> <int>     <int>    <int>
#> 1  2013     1     1       517     2356
#> 2  2013     1     2        42     2354
#> 3  2013     1     3        32     2349
#> 4  2013     1     4        25     2358
#> 5  2013     1     5        14     2357
#> 6  2013     1     6        16     2355
#> # ... with 359 more rows
```

这些函数对筛选操作进行了排秩方面的补充。筛选会返回所有变量，每个观测在单独的一行中：

```
not_cancelled %>%
  group_by(year, month, day) %>%
  mutate(r = min_rank(desc(dep_time))) %>%
  filter(r %in% range(r))
#> Source: local data frame [770 x 20]
#> Groups: year, month, day [365]
#>
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1     2356           2359        -3
#> 3  2013     1     2       42           2359        43
#> 4  2013     1     2     2354           2359        -5
#> 5  2013     1     3       32           2359        33
#> 6  2013     1     3     2349           2359       -10
#> # ... with 764 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>, r <int>
```

计数

你已经见过 `n()`，它不需要任何参数，并返回当前分组的大小。如果想要计算出非缺失值的数量，可以使用 `sum(!is.na(x))`。要想计算出唯一值的数量，可以使用 `n_distinct(x)`：

```
# 哪个目的地具有最多的航空公司?
not_cancelled %>%
  group_by(dest) %>%
  summarize(carriers = n_distinct(carrier)) %>%
  arrange(desc(carriers))
#> # A tibble: 104 × 2
#>    dest carriers
#>   <chr>    <int>
#> 1   ATL        7
```

```
#> 2   BOS        7
#> 3   CLT        7
#> 4   ORD        7
#> 5   TPA        7
#> 6   AUS        6
#> # ... with 98 more rows
```

因为计数太常用了，所以 dplyr 提供了一个简单的辅助函数，用于只需要计数的情况：

```
not_cancelled %>%
  count(dest)
#> # A tibble: 104 × 2
#>    dest     n
#>   <chr> <int>
#> 1   ABQ   254
#> 2   ACK   264
#> 3   ALB   418
#> 4   ANC     8
#> 5   ATL 16837
#> 6   AUS  2411
#> # ... with 98 more rows
```

你还可以选择提供一个加权变量。例如，你可以使用以下代码算出每架飞机飞行的总里程数（实际上就是求和）：

```
not_cancelled %>%
  count(tailnum, wt = distance)
#> # A tibble: 4,037 × 2
#>   tailnum      n
#>     <chr>  <dbl>
#> 1  D942DN   3418
#> 2  N0EGMQ 239143
#> 3  N10156 109664
#> 4  N102UW  25722
#> 5  N103US  24619
#> 6  N104UW  24616
#> # ... with 4,031 more rows
```


逻辑值的计数和比例：`sum(x > 10)` 和 `mean(y == 0)`

当与数值型函数一同使用时，`TRUE` 会转换为 1，`FALSE` 会转换为 0。这使得 `sum()` 和 `mean()` 非常适用于逻辑值：`sum(x)` 可以找出 `x` 中 `TRUE` 的数量，`mean(x)` 则可以找出比例。

```
# 多少架航班是在早上5点前出发的？（这通常表明前一天延误的航班数量）
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(n_early = sum(dep_time < 500))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day n_early
#>   <int> <int> <int>   <int>
#> 1  2013     1     1       0
#> 2  2013     1     2       3
#> 3  2013     1     3       4
```

```
#> 4  2013     1     4       3
#> 5  2013     1     5       3
#> 6  2013     1     6       2
#> # ... with 359 more rows

# 延误超过1小时的航班比例是多少?
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(hour_perc = mean(arr_delay > 60))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day hour_perc
#>   <int> <int> <int>     <dbl>
#> 1  2013     1     1    0.0722
#> 2  2013     1     2    0.0851
#> 3  2013     1     3    0.0567
#> 4  2013     1     4    0.0396
#> 5  2013     1     5    0.0349
#> 6  2013     1     6    0.0470
#> # ... with 359 more rows
```

3.6.5 按多个变量分组

当使用多个变量进行分组时，每次的摘要统计会用掉一个分组变量。这样就可以轻松地对数据集进行循序渐进的分析：

```
daily <- group_by(flights, year, month, day)
(per_day   <- summarize(daily, flights = n()))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day flights
#>   <int> <int> <int>   <int>
#> 1  2013     1     1     842
#> 2  2013     1     2     943
#> 3  2013     1     3     914
#> 4  2013     1     4     915
#> 5  2013     1     5     720
#> 6  2013     1     6     832
#> # ... with 359 more rows

(per_month <- summarize(per_day, flights = sum(flights)))
#> Source: local data frame [12 x 3]
#> Groups: year [?]
#>
#>    year month flights
#>   <int> <int>   <int>
#> 1  2013     1   27004
#> 2  2013     2   24951
#> 3  2013     3   28834
#> 4  2013     4   28330
#> 5  2013     5   28796
```

```
#> 6  2013     6   28243
#> # ... with 6 more rows

(per_year  <- summarize(per_month, flights = sum(flights)))
#> # A tibble: 1 × 2
#>     year flights
#>    <int>   <int>
#> 1  2013  336776
```

在循序渐进地进行摘要分析时，需要小心：使用求和与计数操作是没问题的，但如果想要使用加权平均和方差的话，就要仔细考虑一下，在基于秩的统计数据（如中位数）上是无法进行这些操作的。换句话说，对分组求和的结果再求和就是对整体求和，但分组中位数的中位数可不是整体的中位数。

3.6.6　取消分组

如果想要取消分组，并回到未分组的数据继续操作，那么可以使用 ungroup() 函数：

```
daily %>%
  ungroup() %>%             # 不再按日期分组
  summarize(flights = n())  # 所有航班
#> # A tibble: 1 × 1
#>   flights
#>     <int>
#> 1  336776
```

3.6.7　练习

(1) 通过头脑风暴，至少找出 5 种方法来确定一组航班的典型延误特征。思考以下场景。

- 一架航班 50% 的时间会提前 15 分钟，50% 的时间会延误 15 分钟。
- 一架航班总是会延误 10 分钟。
- 一架航班 50% 的时间会提前 30 分钟，50% 的时间会延误 30 分钟。
- 一架航班 99% 的时间会准时，1% 的时间会延误 2 个小时。

哪一种更重要：到达延误还是出发延误？

(2) 找出另外一种方法，这种方法要可以给出与 not_cancelled %>% count(dest) 和 not_cancelled %>% count(tailnum, wt = distance) 同样的输出（不能使用 count()）。

(3) 我们对已取消航班的定义 (is.na(dep_delay)) | (is.na(arr_delay)) 稍有欠佳。为什么？哪一列才是最重要的？

(4) 查看每天取消的航班数量。其中存在模式吗？已取消航班的比例与平均延误时间有关系吗？

(5) 哪个航空公司的延误情况最严重？挑战：你能否分清这是由于糟糕的机场设备，还是航空公司的问题？为什么能？为什么不能？（提示：考虑一下 flights %>% group_by(carrier, dest) %>% summarize(n())。）

(6) 计算每架飞机在第一次延误超过 1 小时前的飞行次数。

(7) count() 函数中的 sort 参数的作用是什么？何时应该使用这个参数？

3.7 分组新变量（和筛选器）

虽然与 summarize() 函数结合起来使用是最有效的，但分组也可以与 mutate() 和 filter() 函数结合，以完成非常便捷的操作。

- 找出每个分组中最差的成员：

```
flights_sml %>%
  group_by(year, month, day) %>%
  filter(rank(desc(arr_delay)) < 10)
#> Source: local data frame [3,306 x 7]
#> Groups: year, month, day [365]
#>
#>    year month   day dep_delay arr_delay distance
#>   <int> <int> <int>     <dbl>     <dbl>    <dbl>
#> 1  2013     1     1       853       851      184
#> 2  2013     1     1       290       338     1134
#> 3  2013     1     1       260       263      266
#> 4  2013     1     1       157       174      213
#> 5  2013     1     1       216       222      708
#> 6  2013     1     1       255       250      589
#> # ... with 3,300 more rows, and 1 more variables:
#> #   air_time <dbl>
```

- 找出大于某个阈值的所有分组：

```
popular_dests <- flights %>%
  group_by(dest) %>%
  filter(n() > 365)
popular_dests
#> Source: local data frame [332,577 x 19]
#> Groups: dest [77]
#>
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> 5  2013     1     1      554            600        -6
#> 6  2013     1     1      554            558        -4
#> # ... with 3.326e+05 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

- 对数据进行标准化以计算分组指标：

```
popular_dests %>%
  filter(arr_delay > 0) %>%
  mutate(prop_delay = arr_delay / sum(arr_delay)) %>%
  select(year:day, dest, arr_delay, prop_delay)
#> Source: local data frame [131,106 x 6]
#> Groups: dest [77]
#>
#>    year month   day  dest arr_delay prop_delay
#>   <int> <int> <int> <chr>     <dbl>      <dbl>
#> 1  2013     1     1   IAH        11   1.11e-04
#> 2  2013     1     1   IAH        20   2.01e-04
#> 3  2013     1     1   MIA        33   2.35e-04
#> 4  2013     1     1   ORD        12   4.24e-05
#> 5  2013     1     1   FLL        19   9.38e-05
#> 6  2013     1     1   ORD         8   2.83e-05
#> # ... with 1.311e+05 more rows
```

分组筛选器的作用相当于分组新变量加上未分组筛选器。我一般不使用分组筛选器，除非是为了完成快速、粗略的数据处理，否则很难检查数据处理的结果是否正确。

在分组新变量和筛选器中最常使用的函数称为窗口函数（与用于统计的摘要函数相对）。你可以在相应的使用指南中学习到更多关于窗口函数的知识：`vignette("window-functions")`。

练习

(1) 查看常用的新变量函数和筛选函数的列表。当它们与分组操作结合使用时，功能有哪些变化？

(2) 哪一架飞机（用机尾编号来识别，`tailnum`）具有最差的准点记录？

(3) 如果想要尽量避免航班延误，那么应该在一天中的哪个时间搭乘飞机？

(4) 计算每个目的地的延误总时间的分钟数，以及每架航班到每个目的地的延误时间比例。

(5) 延误通常是由临时原因造成的：即使最初引起延误的问题已经解决，但因为要让前面的航班先起飞，所以后面的航班也会延误。使用 `lag()` 函数探究一架航班延误与前一架航班延误之间的关系。

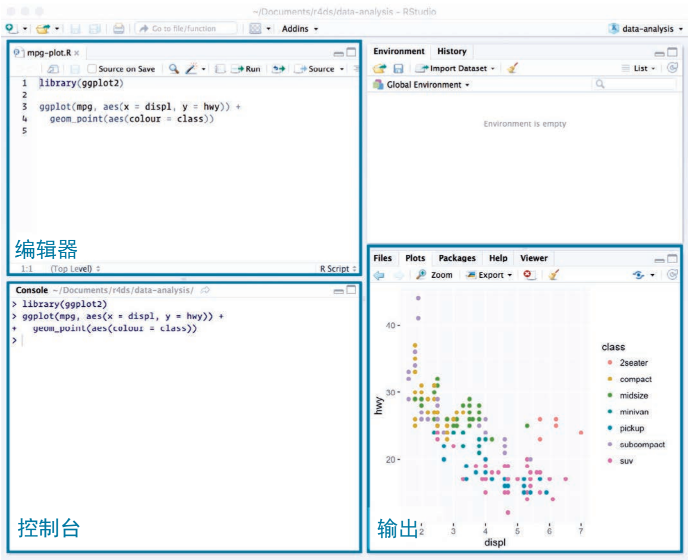
(6) 查看每个目的地。你能否发现有些航班的速度快得可疑？（也就是说，这些航班的数据可能是错误的。）计算出到目的地的最短航线的飞行时间。哪架航班在空中的延误时间最长？

(7) 找出至少有两个航空公司的所有目的地。使用数据集中的信息对航空公司进行排名。

第 4 章

工作流：脚本

迄今为止，我们一直使用 RStudio 控制台来运行代码。这是一个非常好的开始，但如果需要创建更复杂的 ggplot2 图形或者 dplyr 管道，你很快就会发现控制台非常不方便。为了拓展工作空间，我们应该使用 RStudio 脚本编辑器。要想打开脚本编辑器，可以点击 File 菜单，选择 New File，接着选择 R Script；也可以使用组合键 Ctrl+Shift+N。现在你可以看到 4 个窗格。



如果你很重视一段代码，那么脚本编辑器就是存放这段代码的绝好位置。你可以在控制台中不断调试，一旦代码正常运行并输出预期结果，你就可以将其放在脚本编辑器中。当退出 RStudio 时，它会自动保存编辑器中的内容，并在重新打开时自动加载编辑器中的内容。尽管如此，我们还是应该定时保存脚本，并做好备份。

4.1 运行代码

脚本编辑器还非常适合建立复杂的 ggplot2 图形或较长的 dplyr 操作序列。有效使用脚本编辑器的关键是记住最重要的快捷键之一：Ctrl+Enter。这组快捷键会在控制台中执行当前的 R 语句。例如，输入以下代码后，如果光标在█处，那么按 Ctrl+Enter 会运行生成 `not_cancedlled` 的完整命令，并将光标移到下一个语句（即以 `not_cancelled %>%` 开头的语句）。因此，重复按 Ctrl+Enter 就可以轻松运行整个脚本：

```
library(dplyr)
library(nycflights13)

not_cancelled <- flights %>%
  filter(!is.na(dep_delay)█, !is.na(arr_delay))

not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(mean = mean(dep_delay))
```

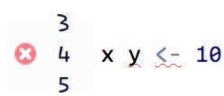
除了按照语句顺序运行，还可以一次性运行整个脚本：Ctrl+Shift+S。定期运行整个脚本是非常好的做法，可以让你确认脚本中所有重要的代码都没有问题。

我们建议你一直将所需要 R 包的语句放在脚本开头。这样一来，如果将代码分享给别人，他们就可以很容易地知道需要安装哪些 R 包。但注意，永远不要在分享的脚本中包括 `install.packages()` 函数或 `setwd()` 函数。这种改变他人计算机设置的行为会引起众怒的。

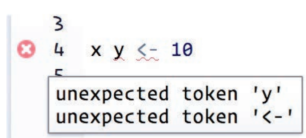
在后续章节的学习中，我们强烈建议你使用脚本编辑器，并练习使用快捷键。随着时间的推移，利用这种方式向控制台发送代码会让人感到得心应手，如同行云流水。

4.2 RStudio自动诊断

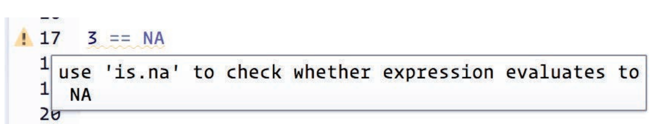
脚本编辑器还会利用红色波浪线和边栏的红叉来高亮显示语法错误：



将鼠标移到红叉上就可以看到错误提示：



RStudio 还能找出潜在的代码问题：



练习

(1) 访问 RStudio 技巧的 twitter 账号 @rstudiotips（https://twitter.com/rstudiotips），选择一个有趣的小技巧，并实践一下！

(2) RStudio 自动诊断程序还会通报哪些其他常见错误？阅读 https://support.rstudio.com/hc/en-us/articles/205753617-Code-Diagnostics 来找出答案。

第 5 章 探索性数据分析

5.1 简介

本章将展示如何使用可视化方法和数据转换来系统化地探索数据，统计学家将这项任务称为探索性数据分析（exploratory data analysis，EDA）。EDA 是一个可迭代的循环过程，具有以下作用。

(1) 对数据提出问题。
(2) 对数据进行可视化、转换和建模，进而找出问题的答案。
(3) 使用上一个步骤的结果来精炼问题，并提出新问题。

EDA 并不是具有严格规则的正式过程，它首先是一种思维状态。在 EDA 的初始阶段，你应该天马行空地发挥想象力，并考察和试验能够想到的所有方法。有些想法是行得通的，有些想法则会无疾而终。当探索更进一步时，你就可以锁定容易产生成果的几个领域，将最终想法整理成文，并与他人进行沟通。

EDA 是所有数据分析过程中的重要环节，因为总是需要考察一下数据质量，即使你可以不费吹灰之力就找出问题。数据清洗只是 EDA 的一项具体应用，此时你提出的问题是，数据是否符合预期。要想进行数据清洗，需要使用所有的 EDA 工具：可视化、数据转换和建模。

准备工作

本章会将你学到的关于 dplyr 和 ggplot2 的知识综合起来，以交互的方式提出问题、用数据解答问题、再提出新的问题。

```
library(tidyverse)
```

5.2 问题

> 没有一成不变的统计问题，统计上的一成不变都是有问题的。
>
> ——Sir David Cox

> 正确问题的近似答案通常是模糊的，但它远远胜过错误问题的确切答案，尽管后者总是很精确。
>
> ——John Tukey

EDA 期间的目标是获取对数据的理解。进行 EDA 的最简单的方式就是将问题作为指导调查研究的工具。提出问题后，这个问题就使得你将注意力集中在数据集中的特定部分，并帮助你进行有关图形、模型和数据转换的决定。

EDA 本质上是一个创造性过程。和多数创造性过程一样，问题的**质量**关键在于问题的**数量**。分析过程的开始阶段很难提出有启发性的问题，因为你并不知道数据集中包含了哪些真知灼见。另一方面，你提出的每个新问题都可以揭示数据中的新内容，并增加发现知识的机会。如果在知识发现的基础上不断使用新问题来补充每个老问题，那么你就可以快速地获取数据中最令人感兴趣的部分，并总结出一组发人深省的问题。

对于应该提出什么样的问题来指导我们的研究，现在还没有确定的规则。但有两类问题总是有助于我们在数据中发现知识。我们可以粗略地将这两类问题表述如下。

(1) 变量本身会发生何种变动？

(2) 不同变量之间会发生何种相关变动？

本章剩余部分会继续讨论这两个问题。我们将解释什么是变动，什么是相关变动，并介绍回答这两个问题的几种方法。为了简化讨论，我们先定义几个术语。

- **变量**：一种可测量的数量、质量或属性。
- **值**：变量在测量时的状态。变量值在每次测量之间可以发生改变。
- **观测**：或称**个案**，指在相同条件下进行的一组测量（通常，一个观测中的所有测量是在同一时间对同一对象进行的）。一个观测会包含多个值，每个值关联到不同的变量。有时我们会将观测称为数据点。
- **表格数据**：一组值的集合，其中每个值都关联一个变量和一个观测。如果每个值都有自己所属的“单元”，每个变量都有自己所属的列，每个观测都有自己所属的行，那么表格数据就是**整洁的**。

迄今为止，我们见过的数据都是整洁的。但在实际生活中，大多数的数据则是不整洁的。

5.3 变动

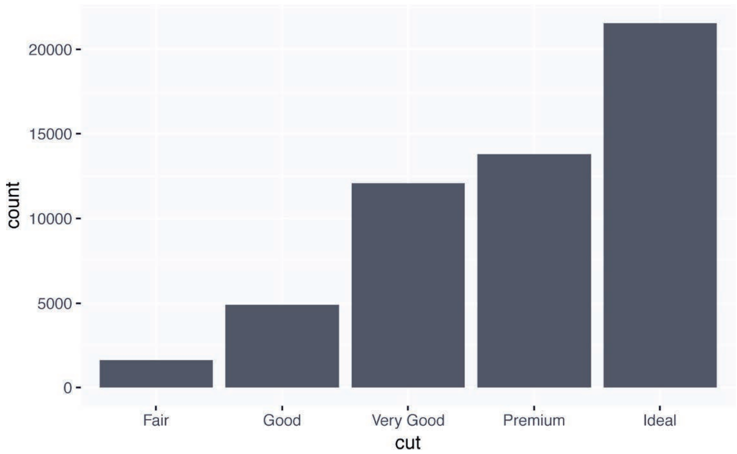
变动是每次测量时数据值的变化趋势。实际生活中很容易看到变动。如果对任意连续型变量进行两次测量，那么会得到两个不同的结果，即使测量的是一个常数（如光速），情况也是如此。每次测量的结果都包括少量误差，误差在每次测量间是不同的。如果测量多个

项目（如不同人的眼睛颜色）或进行多次测量（如电池在不同时刻的电量），分类变量也会发生变化。所有变量都有自己的变动模式，可以揭示出一些有趣的信息。理解这种模式的最好方法就是对变量值的分布进行可视化表示。

5.3.1 对分布进行可视化表示

对变量分布进行可视化的方法取决于变量是分类变量还是连续变量。如果仅在较小的集合内取值，那么这个变量就是分类变量。分类变量在 R 中通常保存为因子或字符向量。要想检查分类变量的分布，可以使用条形图：

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```

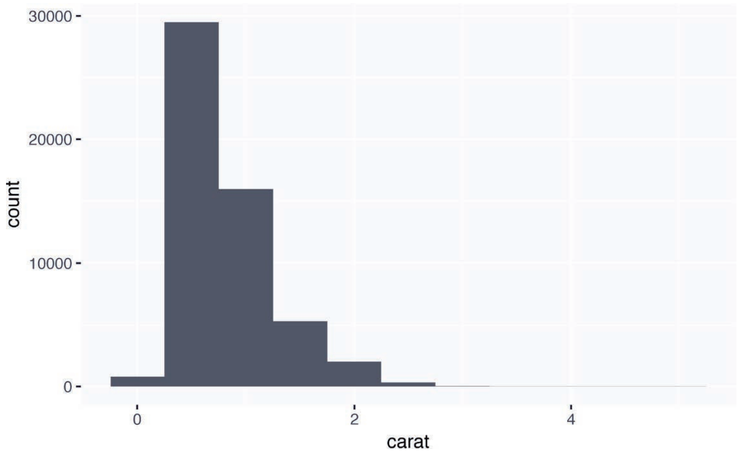


条形的高度表示每个 x 值中观测的数量，你可以使用 `dplyr::count()` 手动计算出这些值：

```
diamonds %>%
  count(cut)
#> # A tibble: 5 × 2
#>         cut     n
#>       <ord> <int>
#> 1      Fair  1610
#> 2      Good  4906
#> 3 Very Good 12082
#> 4   Premium 13791
#> 5     Ideal 21551
```

如果可以在无限大的有序集合中任意取值，那么这个变量就是连续变量。数值型和日期时间型变量就是连续变量的两个例子。要想检查连续变量的分布，可以使用直方图：

```
ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = carat), binwidth = 0.5)
```



你可以通过 dplyr::count() 和 ggplot2::cut_width() 函数的组合来手动计算结果：

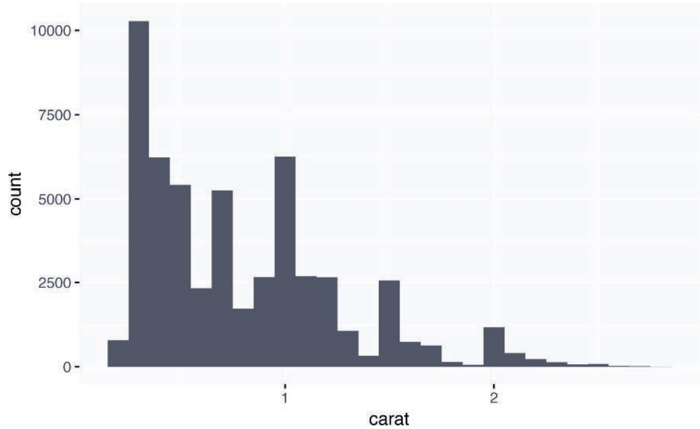
```
diamonds %>%
  count(cut_width(carat, 0.5))
#> # A tibble: 11 × 2
#>   `cut_width(carat, 0.5)`     n
#>                    <fctr> <int>
#> 1             [-0.25,0.25]   785
#> 2              (0.25,0.75] 29498
#> 3              (0.75,1.25] 15977
#> 4              (1.25,1.75]  5313
#> 5              (1.75,2.25]  2002
#> 6              (2.25,2.75]   322
#> # ... with 5 more rows
```

直方图对 x 轴进行等宽分箱，然后使用条形的高度来表示落入每个分箱的观测的数量。在上图中，最高的条形表示几乎有 30 000 个观测的 carat 值在 0.25 和 0.75 之间，这两个值分别是条形的左侧值和右侧值。

你可以使用 binwidth 参数来设定直方图中的间隔的宽度，该参数是用 x 轴变量的单位来度量的。在使用直方图时，你应该试验一下不同的分箱宽度，因为不同的分箱宽度可以揭示不同的模式。例如，如果只考虑重量小于 3 克拉的钻石，并选择一个更小的分箱宽度，那么直方图如下所示：

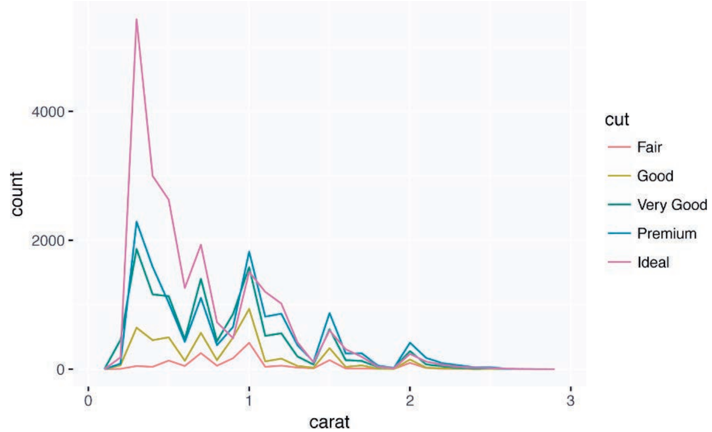
```
smaller <- diamonds %>%
  filter(carat < 3)

ggplot(data = smaller, mapping = aes(x = carat)) +
  geom_histogram(binwidth = 0.1)
```



如果想要在同一张图上叠加多个直方图，那么我们建议你使用 geom_freqploy() 函数来代替 geom_histogram() 函数。geom_freqploy() 可以执行和 geom_histogram() 同样的计算过程，但前者不使用条形来显示计数，而是使用折线。叠加的折线远比叠加的条形更容易理解：

```
ggplot(data = smaller, mapping = aes(x = carat, color = cut)) +
  geom_freqpoly(binwidth = 0.1)
```



这种类型的图存在几个问题，5.5.1 节会对其进行深入讨论。

5.3.2 典型值

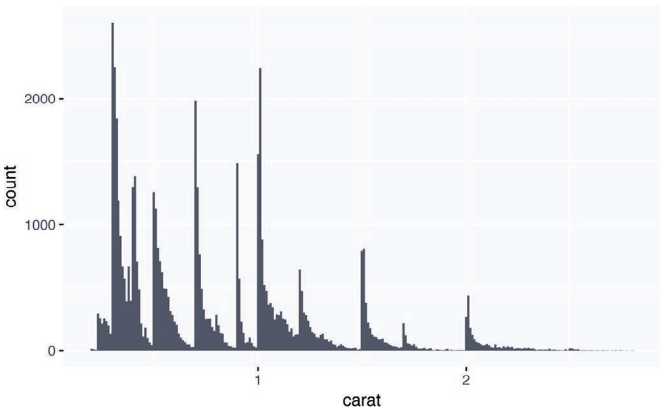
条形图和直方图都用比较高的条形表示变量中的常见值，而用比较矮的条形表示变量中不常见的值。没有条形的位置表示数据中没有这样的值。为了将这些信息转换为有用的问题，我们看看是否具有意料之外的情况。

- 哪些值是最常见的？为什么？
- 哪些值是非常罕见的？为什么？这和你的预期相符吗？
- 你能发现任何异乎寻常的模式吗？如何解释？

作为示例，可以从以下直方图发现几个有趣的问题。

- 为什么重量为整数克拉和常见分数克拉的钻石更多？
- 为什么位于每个峰值稍偏右的钻石比稍偏左的钻石更多？
- 为什么没有重量超过 3 克拉的钻石？

```
ggplot(data = smaller, mapping = aes(x = carat)) +
  geom_histogram(binwidth = 0.01)
```

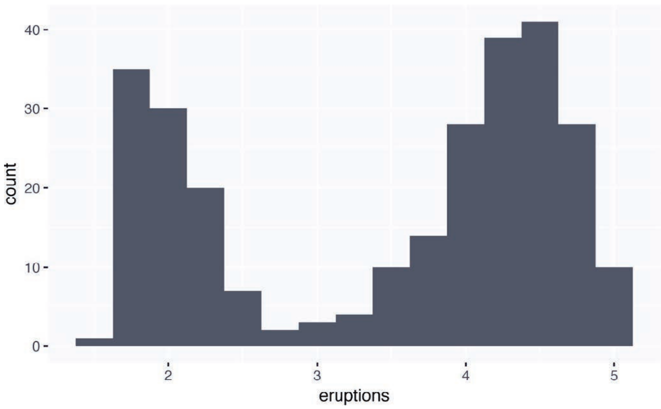


一般来说，相似值聚集形成的簇表示数据中存在子组。为了理解子组，我们提出以下问题。

- 每个簇中的观测是如何相似的？
- 不同簇之间的观测是如何不相似的？
- 如何解释或描述各个簇？
- 为什么有些簇的外观可能具有误导作用？

以下的直方图显示了美国黄石国家公园中的老忠实喷泉的 272 次喷发的时长（单位为分钟）。喷发时间似乎聚集成了两组：短喷发（2 分钟左右）和长喷发（4~5 分钟），这两组间几乎没有其他喷发时间：

```
ggplot(data = faithful, mapping = aes(x = eruptions)) +
  geom_histogram(binwidth = 0.25)
```

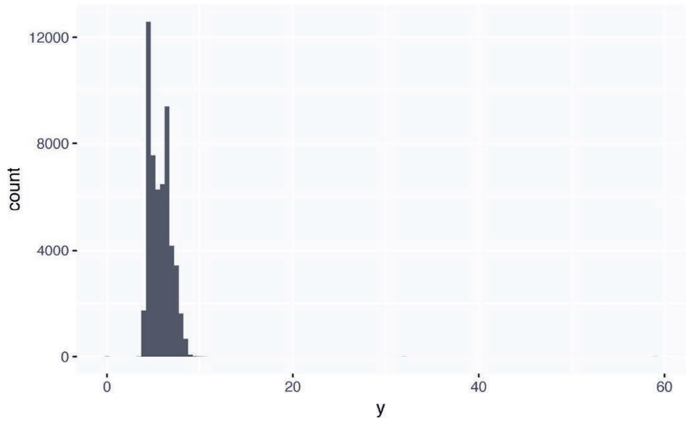


以上很多问题都会促使你探索变量间的关系，例如，查看变量的值是否可以解释另一个变量的行为。我们稍后就会讨论这个问题。

5.3.3 异常值

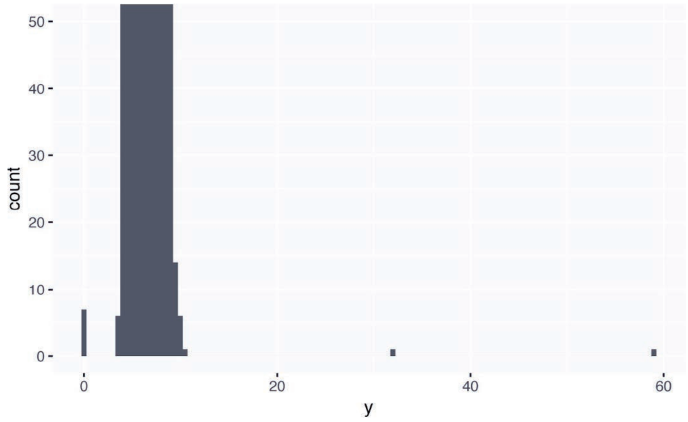
异常值是与众不同的观测或者是模式之外的数据点。有时异常值是由于数据录入错误而产生的；有时异常值则能开辟出一块重要的新科学领域。如果数据量比较大，有时很难在直方图上发现异常值。例如，查看钻石数据集中 y 轴变量的分布，唯一能表示存在异常值的证据是，y 轴的取值范围出奇得宽：

```
ggplot(diamonds) +
  geom_histogram(mapping = aes(x = y), binwidth = 0.5)
```



正常值分箱中的观测太多了，以致于包括异常值的分箱高度太低，因此我们根本看不见（如果仔细观察 x 轴 0 刻度附近，没准你能发现点什么）。为了更容易发现异常值，我们可以使用 `coord_cartesian()` 函数将 y 轴靠近 0 的部分放大：

```
ggplot(diamonds) +
  geom_histogram(mapping = aes(x = y), binwidth = 0.5) +
  coord_cartesian(ylim = c(0, 50))
```



（coord_cartesian() 函数中有一个用于放大 *x* 轴的 xlim() 参数。ggplot2 中也有功能稍有区别的 xlim() 和 ylim() 函数：它们会忽略溢出坐标轴范围的那些数据。）

这样一来，我们就可以看出有 3 个异常值，分别位于 0、30 左右和 60 左右。我们使用 dplyr 将它们找出来：

```
unusual <- diamonds %>%
  filter(y < 3 | y > 20) %>%
  arrange(y)
unusual
#> # A tibble: 9 × 10
#>   carat       cut color clarity depth table price     x
#>   <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int> <dbl>
#> 1  1.00 Very Good     H     VS2  63.3    53  5139  0.00
#> 2  1.14      Fair     G     VS1  57.5    67  6381  0.00
#> 3  1.56     Ideal     G     VS2  62.2    54 12800  0.00
#> 4  1.20   Premium     D    VVS1  62.1    59 15686  0.00
#> 5  2.25   Premium     H     SI2  62.8    59 18034  0.00
#> 6  0.71      Good     F     SI2  64.1    60  2130  0.00
#> 7  0.71      Good     F     SI2  64.1    60  2130  0.00
#> 8  0.51     Ideal     E     VS1  61.8    55  2075  5.15
#> 9  2.00   Premium     H     SI2  58.9    57 12210  8.09
#> # ... with 2 more variables:
#> #   y <dbl>, z <dbl>
```

y 变量测量钻石的三个维度之一，单位为毫米。我们知道钻石的宽度不可能是 0 毫米，因此这些值肯定是错误的。我们也完全可以认为 32 毫米和 59 毫米同样是令人难以置信的，这样的钻石长度超过 1 英寸（1 英寸 =2.54 厘米），简直就是无价之宝！

使用带有异常值和不带异常值的数据分别进行分析，是一种良好的做法。如果两次分析的结果差别不大，而你又无法说明为什么会有异常值，那么完全可以用缺失值替代异常值，然后继续进行分析。但如果两次分析的结果有显著差别，那么你就不能在没有正当理由的情况下丢弃它们。你需要弄清出现异常值的原因（如数据输入错误），并在文章中说明丢弃它们的理由。

5.3.4　练习

(1) 研究 x、y 和 z 变量在 diamonds 数据集中的分布。你能发现什么？思考一下，对于一条钻石数据，如何确定表示长、宽和高的变量？

(2) 研究 price 的分布，你能发现不寻常或令人惊奇的事情吗？（提示：仔细考虑一下 binwidth 参数，并确定试验了足够多的取值。）

(3) 0.99 克拉的钻石有多少？ 1 克拉的钻石有多少？造成这种区别的原因是什么？

(4) 比较并对比 coord_cartesina() 和 xlim()/ylim() 在放大直方图时的功能。如果不设置 binwidth 参数，会发生什么情况？如果将直方图放大到只显示一半的条形，那么又会发生什么情况？

5.4 缺失值

如果在数据集中发现异常值，但只想继续进行其余的分析工作，那么有 2 种选择。

- 将带有可疑值的行全部丢弃：

```
diamonds2 <- diamonds %>%
  filter(between(y, 3, 20))
```

  我们不建议使用这种方式，因为一个无效测量不代表所有测量都是无效的。此外，如果数据质量不高，若对每个变量都采取这种做法，那么你最后可能会发现数据已经所剩无几！

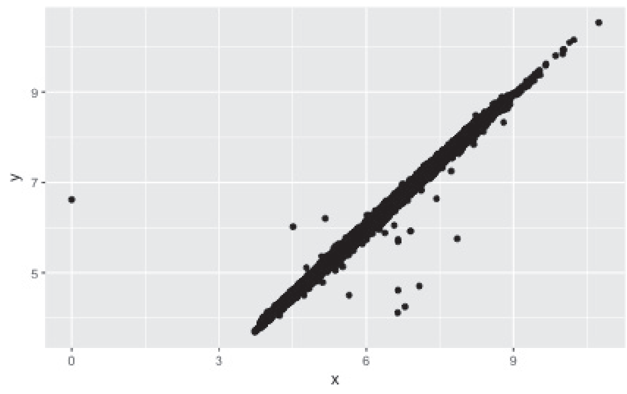
- 相反，我们建议使用缺失值来代替异常值。最简单的做法就是使用 `mutate()` 函数创建一个新变量来代替原来的变量。你可以使用 `ifelse()` 函数将异常值替换为 `NA`：

```
diamonds2 <- diamonds %>%
  mutate(y = ifelse(y < 3 | y > 20, NA, y))
```

`ifelse()` 函数有 3 个参数。第一个参数 `test` 应该是一个逻辑向量，如果 `test` 为 `TRUE`，函数结果就是第二个参数 `yes` 的值；如果 `test` 为 `FALSE`，函数结果就是第三个参数 `no` 的值。

和 R 一样，ggplot2 也遵循不能无视缺失值的原则。因为无法明确地绘制出缺失值，所以 ggplot2 在绘图时会忽略缺失值，但会提出警告以通知缺失值被丢弃了：

```
ggplot(data = diamonds2, mapping = aes(x = x, y = y)) +
  geom_point()
#> Warning: Removed 9 rows containing missing values
#> (geom_point).
```

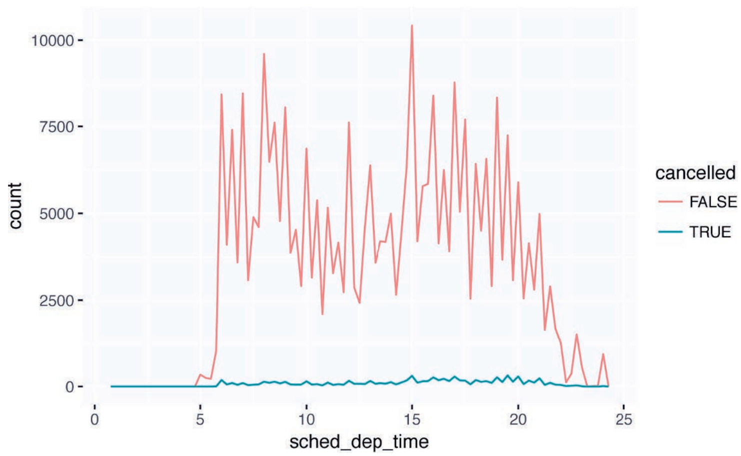


要想不显示这条警告，可以设置 `na.rm = TRUE`：

```
ggplot(data = diamonds2, mapping = aes(x = x, y = y)) +
  geom_point(na.rm = TRUE)
```

有时你会想弄清楚造成有缺失值的观测和没有缺失值的观测间的区别的原因。例如，在 `nycflights13::flights` 中，`dep_time` 变量中的缺失值表示航班取消了。因此，你应该比较一下已取消航班和未取消航班的计划出发时间。可以使用 `is.na()` 函数创建一个新变量来完成这个操作：

```
nycflights13::flights %>%
  mutate(
    cancelled = is.na(dep_time),
    sched_hour = sched_dep_time %/% 100,
    sched_min = sched_dep_time %% 100,
    sched_dep_time = sched_hour + sched_min / 60
  ) %>%
  ggplot(mapping = aes(sched_dep_time)) +
    geom_freqpoly(
      mapping = aes(color = cancelled),
      binwidth = 1/4
    )
```



但是这张图的效果并不怎么好，因为未取消航班的数量远远多于已取消航班的数量。我们会在下一节中介绍改善此类比较的一些技巧。

练习

(1) 直方图如何处理缺失值？条形图如何处理缺失值？为什么会有这种区别？

(2) `na.rm = TRUE` 在 `mean()` 和 `sum()` 函数中的作用是什么？

5.5　相关变动

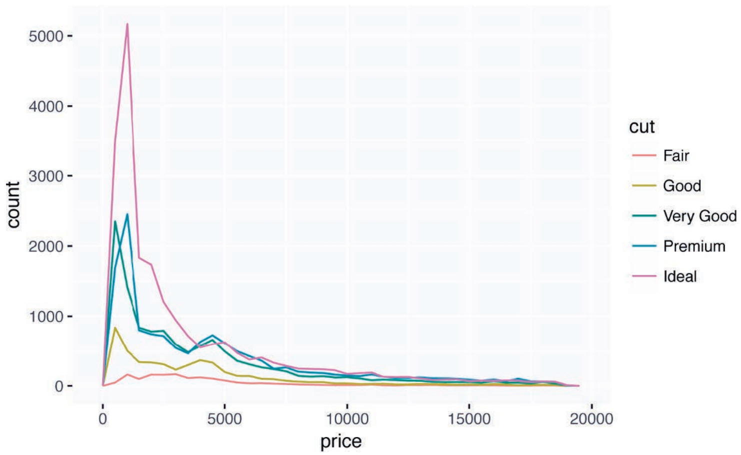
如果变动描述的是一个变量**内部**的行为，那么相关变动描述的就是多个变量**之间**的行为。**相关变动**是两个或多个变量以相关的方式共同变化所表现出的趋势。查看相关变动的最好方式是将两个或多个变量间的关系以可视化的方式表现出来。如何进行这种可视化表示同样取决于相关变量的类型。

5.5.1　分类变量与连续变量

我们经常需要探索连续变量的分布，这种分布按照一个分类变量的值可以分为几个组，就像前面的频率多边形图一样。`geom_freqpoly()` 的默认外观不太适合这种比较，因为高度是

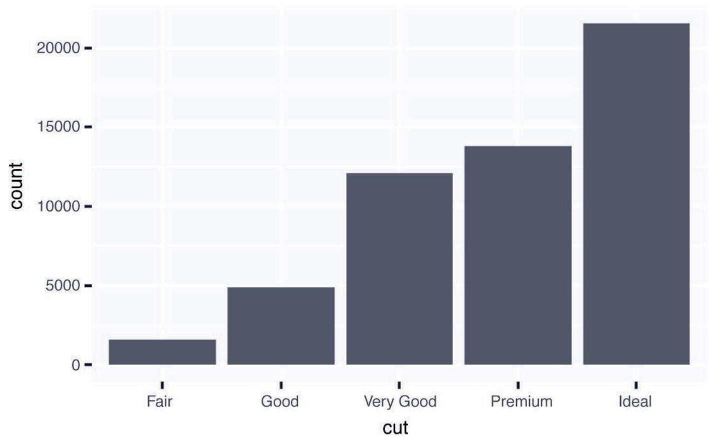
由计数给出的。这就意味着，如果一组观测的数量明显少于其他组的话，就很难看出形状上的差别。举个例子，我们探索一下钻石价格是如何随着质量而变化的：

```
ggplot(data = diamonds, mapping = aes(x = price)) +
  geom_freqpoly(mapping = aes(color = cut), binwidth = 500)
```



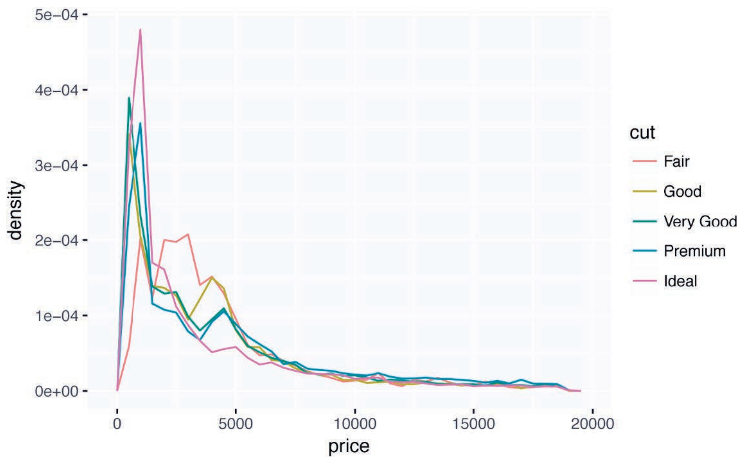
很难看出分布上的差别，因为总体看来各组数量的差别太大了：

```
ggplot(diamonds) +
  geom_bar(mapping = aes(x = cut))
```



为了让比较变得更容易，需要改变 *y* 轴的显示内容，不再显示计数，而是显示**密度**。密度是对计数的标准化，这样每个频率多边形下边的面积都是 1：

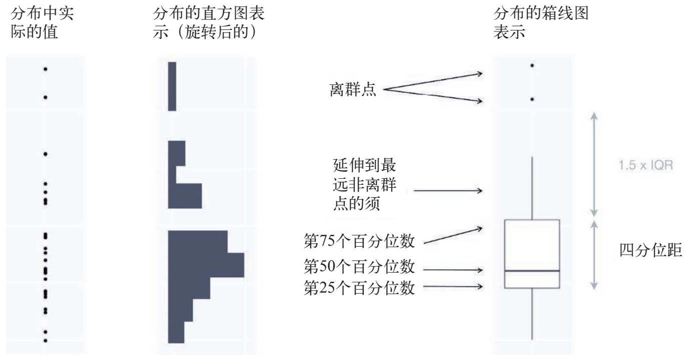
```
ggplot(
  data = diamonds,
    mapping = aes(x = price, y = ..density..)
  ) +
   geom_freqpoly(mapping = aes(color = cut), binwidth = 500)
```



这张图的部分内容非常令人惊讶，其显示出一般钻石（质量最差）的平均价格是最高的！但这可能是因为频率多边形图很难解释，所以这张图还有很多可以改进的地方。

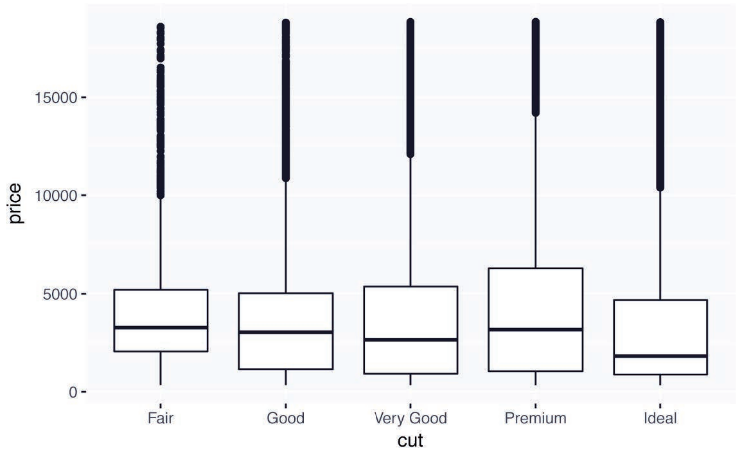
按分类变量的分组显示连续变量分布的另一种方式是使用箱线图。**箱线图**是对变量值分布的一种简单可视化表示，这种图在统计学家中非常流行。每张箱线图都包括以下内容。

- 一个长方形箱子，下面的边表示分布的第 25 个百分位数，上面的边表示分布的第 75 个百分位数，上下两边的距离称为四分位距。箱子的中部有一条横线，表示分布的中位数，也就是分布的第 50 个百分位数。这三条线可以表示分布的分散情况，还可以帮助我们明确数据是关于中位数对称的，还是偏向某一侧。
- 圆点表示落在箱子上下两边 1.5 倍四分位距外的观测，这些离群点就是异常值，因此需要单独绘出。
- 从箱子上下两边延伸出的直线（或称为须）可以到达分布中最远的非离群点处。



使用 `geom_boxplot()` 函数查看按切割质量分类的价格分布：

```
ggplot(data = diamonds, mapping = aes(x = cut, y = price)) +
  geom_boxplot()
```

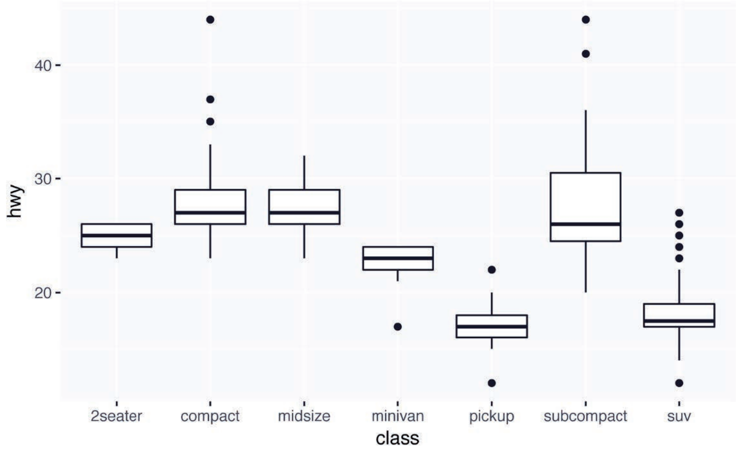


虽然看不出太多关于分布的信息，但箱线图更加紧凑，因此可以更容易地比较多个类别（也更适合使用一张图来表示）。与前面的图形一样，我们可以从箱线图中发现违反直觉的现象：质量更好的钻石的平均价格更低！你将在练习中接受这一挑战，说明为什么会这样。

cut 是一个有序因子：“一般”不如“较好”、“较好”不如“很好”，以此类推。因为很多分类变量并没有这种内在的顺序，所以有时需要对其重新排序来绘制信息更丰富的图形。重新排序的其中一种方法是使用 reorder() 函数。

例如，我们看一下 mpg 数据集中的 class 变量。你可能很想知道公路里程因汽车类别的不同会有怎样的变化：

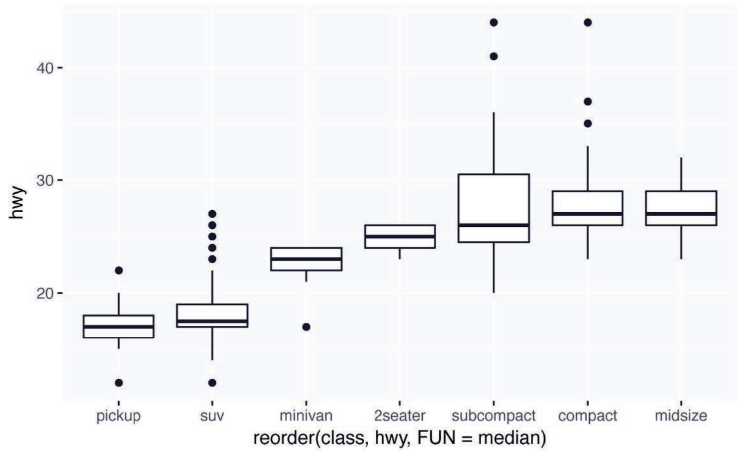
```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot()
```



为了更容易发现趋势，可以基于 hwy 值的中位数对 class 进行重新排序：

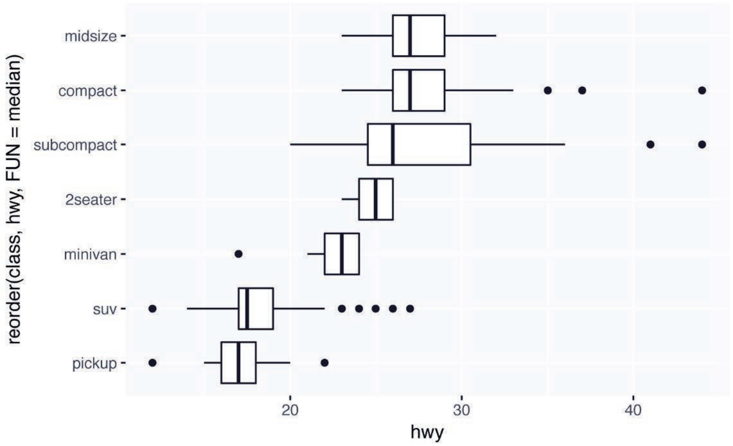
```
ggplot(data = mpg) +
  geom_boxplot(
```

```
    mapping = aes(
      x = reorder(class, hwy, FUN = median),
      y = hwy
    )
  )
```



如果变量名很长，那么将图形旋转 90 度效果会更好一些。你可以通过 `coord_flip()` 函数完成这一操作：

```
ggplot(data = mpg) +
  geom_boxplot(
    mapping = aes(
      x = reorder(class, hwy, FUN = median),
      y = hwy
    )
  ) +
  coord_flip()
```



练习

(1) 前面对比了已取消航班和未取消航班的出发时间，使用学习到的知识对这个对比的可视化结果进行改善。

(2) 在钻石数据集中，哪个变量对于预测钻石的价格最重要？这个变量与切割质量的关系是怎样的？为什么这两个变量的关系组合会导致质量更差的钻石价格更高呢？

(3) 安装 ggstance 包，并创建一个横向箱线图。这种方法与使用 `coord_flip()` 函数有何区别？

(4) 箱线图存在的问题是，在小数据集时代开发而成，对于现在的大数据集会显示出数量极其庞大的异常值。解决这个问题的一种方法是使用字母价值图。安装 lvplot 包，并尝试使用 `geom_lv()` 函数来显示价格基于切割质量的分布。你能发现什么问题？如何解释这种图形？

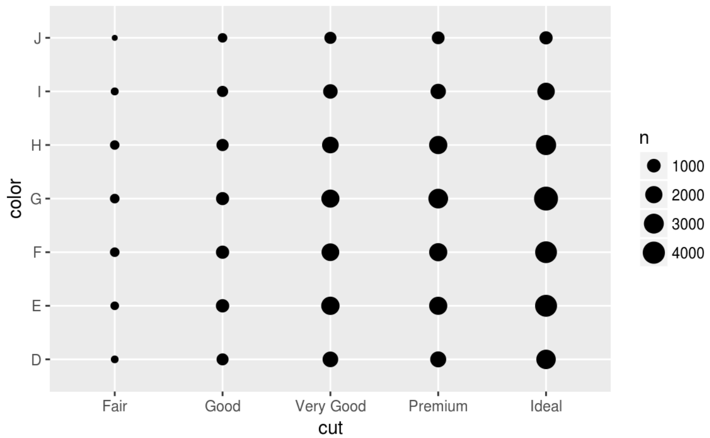
(5) 比较并对比 `geom_violin()`、分面的 `geom_histogram()` 和着色的 `geom_freqploy()`。每种方法的优缺点是什么？

(6) 对于小数据集，如果要观察连续变量和分类变量间的关系，有时使用 `geom_jitter()` 函数是特别有用的。ggbeeswarm 包提供了和 `geom_jitter()` 相似的一些方法。列出这些方法并简单描述每种方法的作用。

5.5.2 两个分类变量

要想对两个分类变量间的相关变动进行可视化表示，需要计算出每个变量组合中的观测数量。完成这个任务的其中一种方法是使用内置的 `geom_count()` 函数：

```
ggplot(data = diamonds) +
  geom_count(mapping = aes(x = cut, y = color))
```



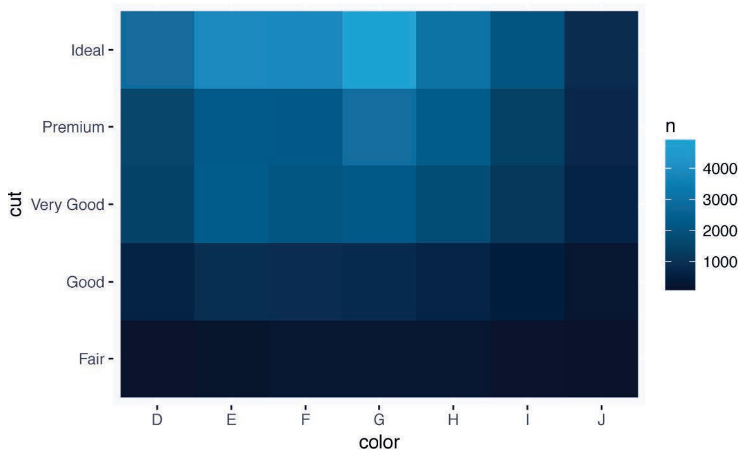
图中每个圆点的大小表示每个变量组合中的观测数量。相关变动就表示为特定 x 轴变量值与特定 y 轴变量值之间的强相关关系。

计算变量组合中的观测数量的另一种方法是使用 dplyr：

```
diamonds %>%
  count(color, cut)
#> Source: local data frame [35 x 3]
#> Groups: color [?]
#>
#>   color       cut     n
#>   <ord>     <ord> <int>
#> 1     D      Fair   163
#> 2     D      Good   662
#> 3     D Very Good  1513
#> 4     D   Premium  1603
#> 5     D     Ideal  2834
#> 6     E      Fair   224
#> # ... with 29 more rows
```

接着使用 geom_tile() 函数和填充图形属性进行可视化表示：

```
diamonds %>%
  count(color, cut) %>%
  ggplot(mapping = aes(x = color, y = cut)) +
    geom_tile(mapping = aes(fill = n))
```



如果分类变量是无序的，那么可以使用 seriation 包对行和列同时进行重新排序，以便更清楚地表示出有趣的模式。对于更大的图形，你可以使用 d3heatmap 或 heatmaply 包，这两个包都可以生成有交互功能的图形。

练习

(1) 如何调整 count 数据，使其能更清楚地表示出切割质量在颜色间的分布，或者颜色在切割质量间的分布？

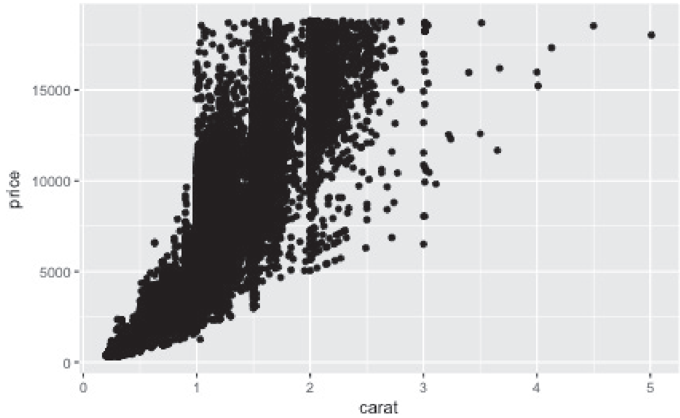
(2) 使用 geom_tile() 函数结合 dplyr 来探索平均航班延误数量是如何随着目的地和月份的变化而变化的。为什么这张图难以阅读？如何改进？

(3) 为什么在以上示例中使用 aes(x = color, y = cut) 要比 aes(x = cut, y = color) 更好？

5.5.3 两个连续变量

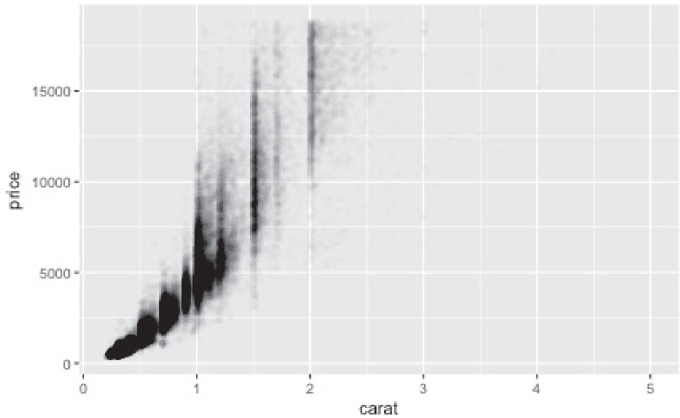
对于两个连续变量间的相关变动的可视化表示，我们已经介绍了一种非常好的方法：使用 geom_point() 画出散点图。你可以将相关变动看作点的模式。例如，你可以看到钻石的克拉数和价值之间存在一种指数关系：

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price))
```



随着数据集规模的不断增加，散点图的用处越来越小，因为数据点开始出现过绘制，并堆积在一片黑色区域中（如上面的散点图所示）。我们已经介绍了解决这个问题的一种方法，即使用 alpha 图形属性添加透明度：

```
ggplot(data = diamonds) +
  geom_point(
    mapping = aes(x = carat, y = price),
    alpha = 1 / 100
  )
```



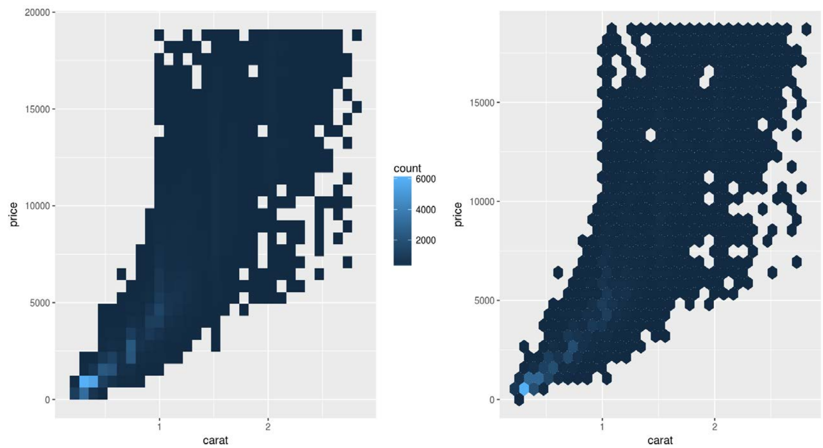
但是很难对特别大的数据集使用透明度。另一种解决方法是使用分箱。我们之前使用了 geom_histogram() 和 geom_freqpoly() 函数在一个维度上进行分箱，现在学习如何使用

geom_bin2d() 和 geom_hex() 函数在两个维度上进行分箱。

geom_bin2d() 和 geom_hex() 函数将坐标平面分为二维分箱，并使用一种填充颜色表示落入每个分箱的数据点。geom_bin2d() 创建长方形分箱。geom_hex() 创建六边形分箱。要想使用 geom_hex()，需要安装 hexbin 包：

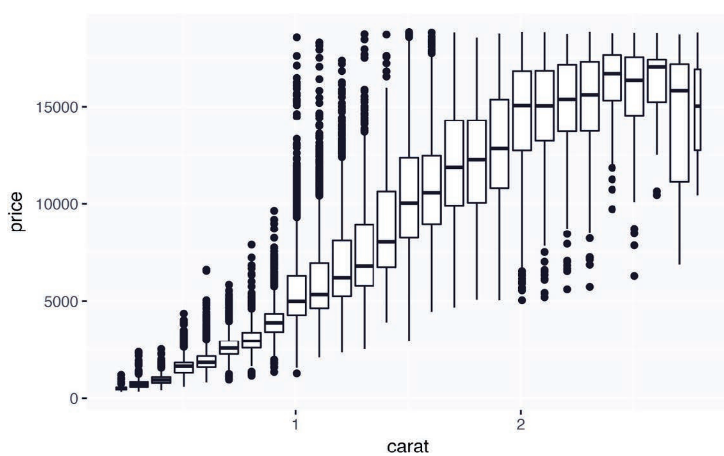
```
ggplot(data = smaller) +
  geom_bin2d(mapping = aes(x = carat, y = price))

# install.packages("hexbin")
ggplot(data = smaller) +
  geom_hex(mapping = aes(x = carat, y = price))
```



另一种方式是对一个连续变量进行分箱，因此这个连续变量的作用就相当于分类变量。接下来就可以使用前面学过的对分类变量和连续变量的组合进行可视化的技术了。例如，你可以对 carat 进行分箱，然后为每个组生成一个箱线图：

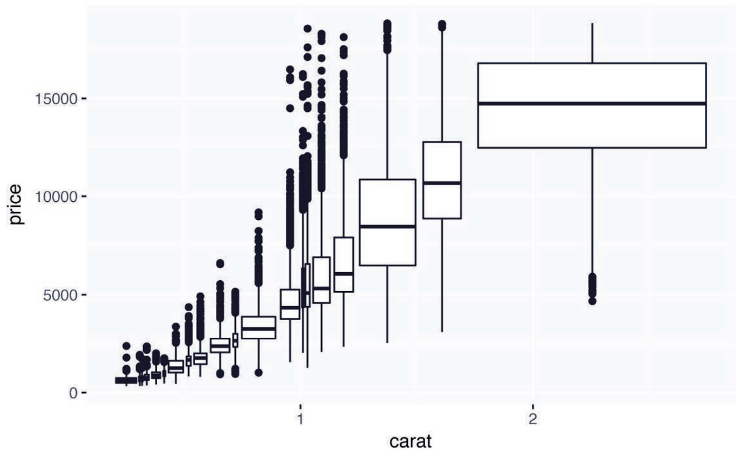
```
ggplot(data = smaller, mapping = aes(x = carat, y = price)) +
  geom_boxplot(mapping = aes(group = cut_width(carat, 0.1)))
```



以上示例使用了 cut_width(x, width) 函数将 x 变量分成宽度为 width 的分箱。默认情况下，不管其中有多少个观测，箱线图看上去都差不多（除了离群点的数量不同），因此很难分辨出每个箱线图是对不同数量的观测进行摘要统计的。如果想要体现这种信息，可以使用参数 varwidth = TRUE 让箱线图的宽度与观测数量成正比。

另一种方法是近似地显示每个分箱中的数据点的数量，此时可以使用 cut_number() 函数：

```
ggplot(data = smaller, mapping = aes(x = carat, y = price)) +
  geom_boxplot(mapping = aes(group = cut_number(carat, 20)))
```



练习

(1) 除了使用箱线图对条件分布进行摘要统计，你还可以使用频率多边形图。使用 cut_width() 函数或 cut_number() 函数时需要考虑什么问题？这对 carat 和 price 的二维分布的可视化表示有什么影响？

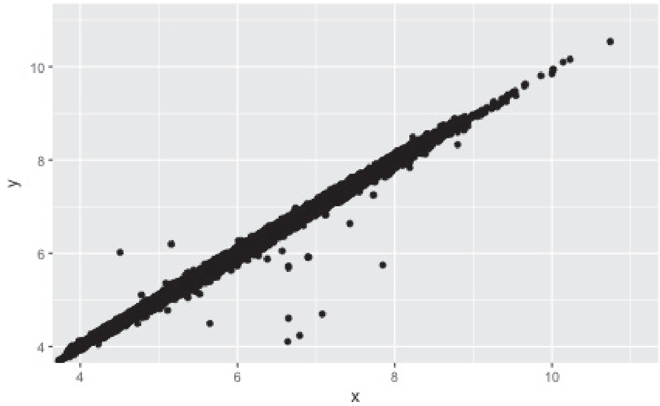
(2) 按照 price 分类对 carat 的分布进行可视化表示。

(3) 比较特别大的钻石和比较小的钻石的价格分布。结果符合预期吗？还是出乎意料？

(4) 组合使用你学习到的两种技术，对 cut、carat 和 price 的组合分布进行可视化表示。

(5) 二维图形可以显示一维图形中看不到的离群点。例如，以下图形中的有些点具有异常的 x 值和 y 值组合，这使得这些点成为了离群点，即使这些点的 x 值和 y 值在单独检验时似乎是正常的。

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = x, y = y)) +
  coord_cartesian(xlim = c(4, 11), ylim = c(4, 11))
```



这种情况下，为什么散点图的显示效果比分箱图更好？

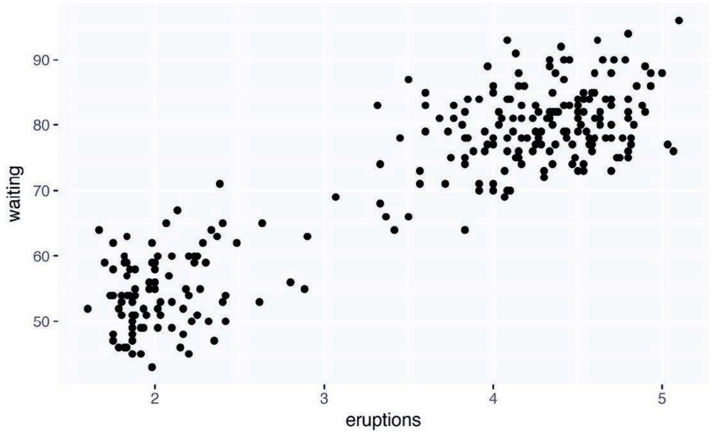
5.6　模式和模型

数据中的模式提供了关系线索。如果两个变量之间存在系统性的关系，那么这种关系就会在数据中表现为一种模式。如果发现了模式，需要问自己以下几个问题。

- 这种模式的出现会不会是一种巧合（也就是随机的偶然因素）？
- 应该如何描述这种模式中隐含的关系？
- 这种模式中隐含的关系有多强？
- 其他变量会如何影响这种关系？
- 如果对数据的独立分组进行检查，这种关系会有所变化吗？

我们就前面提到的美国黄石国家公园中老忠实喷泉的喷发时长和两次喷发之间的等待时间做出一张散点图，该图会显示出一个模式：较长的等待时间与较长的喷发时间是相关的。图中还显示出两个簇，这个我们之前就发现了：

```
ggplot(data = faithful) +
  geom_point(mapping = aes(x = eruptions, y = waiting))
```



模式是数据科学中最有效的工具之一，因为其可以揭示相关变动。如果说变动会生成不确定性，那么相关变动就是减少不确定性。如果两个变量是共同变化的，就可以使用一个变量的值来更好地预测另一个变量的值。如果相关变动可以归因于一种因果关系（一种特殊情况），那么就可以使用一个变量的值来控制另一个变量的值。

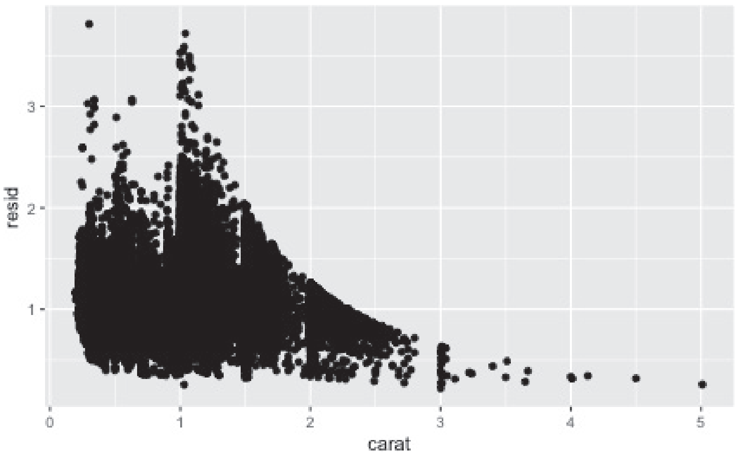
模型是用于从数据中抽取模式的一种工具。例如，我们思考一下钻石数据。切割质量与价格之间的关系是很难理解的，因为切割质量和克拉数以及克拉数和价格之间是紧密相关的。我们可以使用模型去除价格和克拉数之间的强关系，这样就可以继续研究剩余的微妙关系。以下代码拟合了一个模型，可以根据 carat 预测 price，并计算出残差（预测值和实际值之间的差别）。一旦去除克拉数对价格的影响，残差就能反映出钻石的价格：

```
library(modelr)

mod <- lm(log(price) ~ log(carat), data = diamonds)

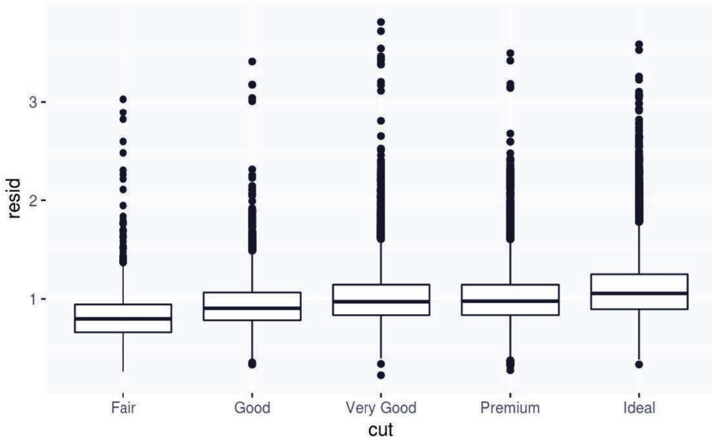
diamonds2 <- diamonds %>%
  add_residuals(mod) %>%
  mutate(resid = exp(resid))

ggplot(data = diamonds2) +
  geom_point(mapping = aes(x = carat, y = resid))
```



去除克拉数和价格之间的强关系后，就可以看到预料中的切割质量与价格的关系，对于同样大小的钻石，切割质量更好的钻石更昂贵：

```
ggplot(data = diamonds2) +
  geom_boxplot(mapping = aes(x = cut, y = resid))
```



我们会在第四部分中介绍模型和 modelr 包的使用方法。之所以将建模保留到后面介绍，是因为一旦掌握了数据处理和编程的工具，理解模型的意义和工作原理就不费吹灰之力了。

5.7 ggplot2调用

结束这些入门章节继续学习前，我们还要介绍 ggplot2 代码的一种更精简表示。迄今为止，ggplot2 的代码已经非常明确，有助于我们的学习。

```
ggplot(data = faithful, mapping = aes(x = eruptions)) +
  geom_freqpoly(binwidth = 0.25)
```

通常情况下，一个函数的前一个或前两个参数是非常重要的，你应该将它们牢记于心。`ggplot()` 函数的前两个参数是 `data` 和 `mapping`，`aes()` 函数的前两个参数是 `x` 和 `y`。在本书剩余的部分中，我们不再写出这些参数名，这样既可以节省输入时间，也可以让代码样板更精简，以便更容易找出两张图之间的不同之处。这是非常重要的编程注意事项，我们还会在第 14 章中进行这方面的讨论。

以更精简的方式重写前面的绘图语句，结果如下所示：

```
ggplot(faithful, aes(eruptions)) +
  geom_freqpoly(binwidth = 0.25)
```

有时我们会将数据转换管道操作的最终结果转换为一张图。注意，此时转换是用 `+` 号实现的，不是 `%>%`。我也不希望如此，但遗憾的是，ggplot2 是在管道操作方式发明前开发出来的。

```
diamonds %>%
  count(cut, clarity) %>%
  ggplot(aes(clarity, cut, fill = n)) +
    geom_tile()
```

5.8 更多学习资源

如果想要学习更多关于 ggplot2 内部机制的内容，我们强烈推荐你买一本 ggplot2 参考书（http://ggplot2.org/book/）。这本书最近进行了更新，因此其中也包括了 dplyr 的代码，还提供了更多内容，让你可以探索数据可视化的方方面面。遗憾的是，这本书不太容易免费获取，但如果可以在大学内上网的话，那么你可以从 SpringerLink 获得免费的电子版。

另一项非常有用的资源是 Winston Chang 所著的 *R Graphic Cookbook*，其中多数内容可以在线获得。

我们还推荐 Antony Unwin 所著的 *Graphical Data Analysis with R*。这本书的内容安排和本章差不多，但讨论的深度远远超过了本章。

第 6 章

工作流：项目

总有一天你会退出 R，做些别的事情，第二天再回过头来继续完成分析工作。总有一天你会使用 R 同时进行多个分析工作，并希望分门别类地保存这些工作。总有一天你会需要从外部世界将数据导入 R，并将数值结果和图表从 R 返回到现实世界。为了解决这些实际生活中的问题，你需要做出以下两个决策。

(1) 分析中的哪些部分是“真实的”？也就是说，你会将哪些部分保存下来作为持久的记录？

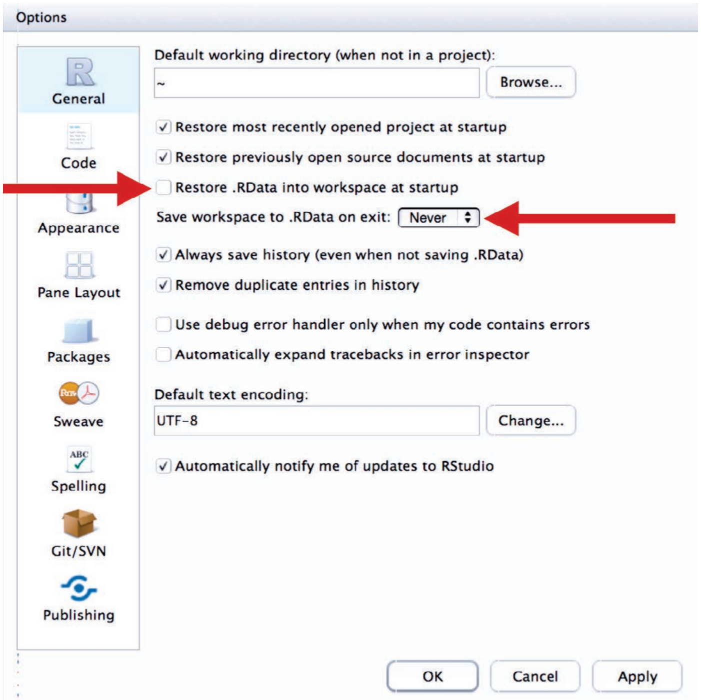
(2) 你的分析工作“位于”哪里？

6.1 什么是真实的

作为 R 的一名新手，你可以认为自己的 R 环境（也就是环境窗格中列出的那些对象）是“真实的”。但从长远来看，你最好认为 R 脚本是“真实的”。

可以通过 R 脚本（以及数据文件）重建 R 环境，但在 R 环境中重建 R 脚本就要困难得多！要么被迫重敲一次内存中的代码（伴随着各种输入错误），要么被迫在 R 历史记录中埋头翻找。

为了培养良好的使用习惯，我们强烈建议你指示 RStudio 不在两次会话间保存工作空间。



这样做会让你短期内有些难受，因为 RStudio 在重新启动时无法记住前一次运行代码的结果。但长痛不如短痛，这样做会迫使你将所有重要的交互都写在代码中。如果只是将一次重要计算的结果保存在工作空间中，而不是将计算过程本身保存在代码中，那么 3 个月后你会发现人世间最痛苦的事莫过于此。

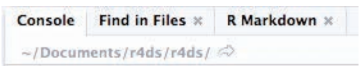
组合使用以下快捷键可以确保重要的代码都写在了编辑器中。

- 按 Ctrl+Shift+F10 来重启 RStudio。
- 按 Ctrl+Shift+S 来重新运行当前脚本。

我们每周会重复这种工作模式几百次。

6.2 你的分析位于哪里

R 中有个名为**工作目录**的重要概念。R 在这个目录中查找你要加载的文件，也将你要保存的文件放在这个目录中。RStudio 在控制台上方显示当前工作目录。



可以通过运行 `getwd()` 命令在 R 代码中输出这个目录：

```
getwd()
#> [1] "/Users/hadley/Documents/r4ds/r4ds"
```

作为 R 语言新手，你可以使用自己的主目录、文档目录或计算机上其他稀奇古怪的目录作

为 R 的工作目录。但既然已经学习了本书的 6 章内容，你也应该掌握一定的知识了。从现在开始，你应该逐渐学会使用目录来组织分析项目，每开始一个项目，就应该将 R 的工作目录设置为与这个项目相关的目录。

还可以使用 R 的命令来设置工作目录，但我们**不建议使用这种方法**：

```
setwd("/path/to/my/CoolProject")
```

不要使用这种操作，因为还有更好的方法，可以让你像专家一样管理与 R 相关的工作。

6.3 路径与目录

路径与目录稍微有一点复杂，因为路径有 2 种基本风格：Mac/Linux 和 Windows。它们主要有以下 3 种区别。

- 最重要的区别是如何分隔路径中的各个部分。Mac 和 Linux 使用的是斜杠（如 `plots/diamonds.pdf`），Windows 使用的则是反斜杠（如 `plots\diamonds.pdf`）。R 支持任意一种类型（不管你现在使用的是哪种平台），但问题是，反斜杠在 R 中具有特殊意义，因此，如果想要表示路径中的单个反斜杠，你需要输入 2 个反斜杠！这有点令人沮丧，因此我们建议你一直使用 Linux/Mac 风格的斜杠。
- 绝对路径（即不管你的工作目录是什么，都指向一个位置的路径）的形式不同。在 Windows 系统中，绝对路径的开头是驱动器号（如 `C:`）或两个反斜杠（如 `\\servername`）；在 Mac/Linux 系统中，绝对路径的开头则是斜杠“/”（如 `/user/hadley`）。**千万不要**在脚本中使用绝对路径，因为不利于分享：没有任何人会和你具有完全相同的目录设置。
- 最后一个小区别是 ~ 指向的位置。~ 是指向主目录的一个很方便的快捷方式。Windows 其实没有主目录的概念，因此 ~ 指向的是文档目录。

6.4 RStudio项目

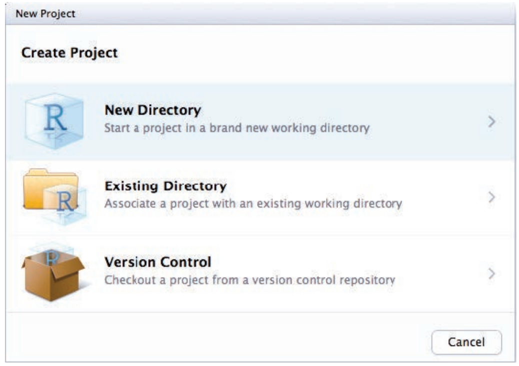
R 专家将与项目相关的所有文件放在一起，其中包括输入数据、R 脚本、分析结果以及图形。因为这是极其明智而又通用的做法，所以 RStudio 通过**项目**对这种做法提供了内置的支持。

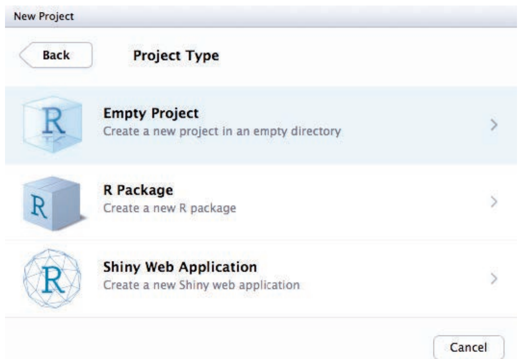
我们建立一个项目以供你在学习本书剩余内容时使用。点击 File → New Project，接下来，将项目命名为 `r4ds`，然后仔细思索一下将项目放在哪个子目录中。如果不将项目放在合适的地方，将来就很难找到它了！

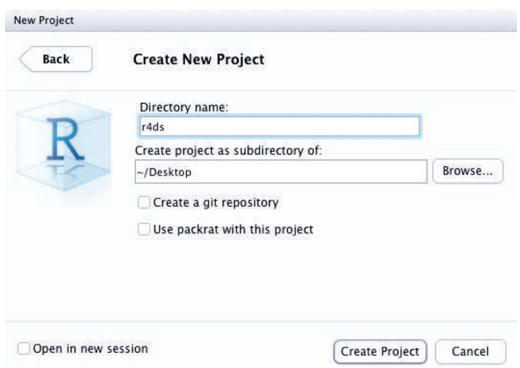
一旦完成这个过程，你就为本书建立了一个新的 RStudio 项目。检查你的项目的“主”目录是否为当前工作目录：

```
getwd()
#> [1] /Users/hadley/Documents/r4ds/r4ds
```

只要使用相对路径引用文件，R 就会在这个目录中寻找文件。







接下来在脚本编辑器中输入以下命令，并保存这个文件，文件名为 diamonds.R。下一步是运行整个脚本，将一个 PDF 文件和一个 CSV 文件保存到项目目录中。不用担心代码中的细节，你会在本书后面学到。

```
library(tidyverse)

ggplot(diamonds, aes(carat, price)) +
  geom_hex()
ggsave("diamonds.pdf")

write_csv(diamonds, "diamonds.csv")
```

退出 RStudio，查看项目目录，你会发现一个 .Rproj 文件。双击这个文件来重新打开项目。注意，你又回到了离开的地方：同样的工作目录，同样的命令历史，你使用过的所有文件都是打开的。因为按照上面的指示进行了操作，所以你会有一个全新的环境，以确保可以从头开始。

使用最喜欢的操作系统方式在计算机中搜索 diamonds.pdf，你不但会找到 PDF 文件（理应如此），还会找到**创建这个文件的脚本**（diamonds.r）。这是巨大的胜利！总有一天你会想重新生成图形，或者想知道图形的来源。如果从来不使用鼠标和剪贴板，而是严格使用 **R 代码**将图形保存到文件的话，你就可以轻松重现以前的工作！

6.5 小结

总的来说，RStudio 项目可以为你提供一个坚实可靠的工作流程，对你未来的工作大有裨益。

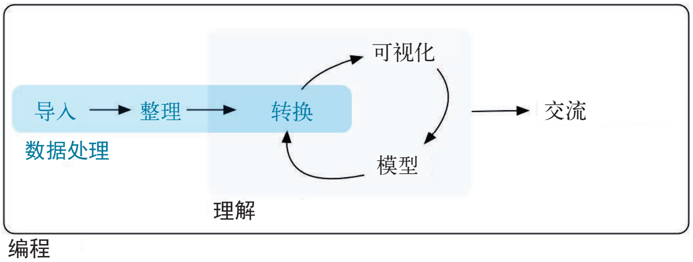
- 为每个数据分析任务创建 RStudio 项目。
- 在项目中保存数据文件。我们会在第 8 章中讨论将数据文件加载到 R 中的方法。
- 在项目中保存脚本。编辑脚本并按照命令运行脚本或运行整个脚本。
- 在项目中保存输出（图形和清洁的数据）。
- 只使用相对路径，不要使用绝对路径。

这样一来，需要的所有文件都放在一处了，而且与所有其他项目干净地隔离开了。

第二部分

数据处理

我们将在这一部分介绍数据处理。数据处理是一门艺术，将数据以合适的形式导入 R，从而进行可视化和建模。数据处理非常重要：没有这个过程，就无法使用数据进行工作！数据处理包括 3 个重要环节。



这一部分的内容安排如下。

- 第 7 章将介绍数据框的一种变体：tibble。我们会在本书中使用这种数据结构。你将会了解 tibble 和普通数据框的区别，以及如何“手工”构造 tibble。
- 第 8 章将介绍如何从磁盘读取数据并导入 R。我们重点关注矩形格式的纯文本文件，但会给出一些指示，表明哪些包可以处理其他类型的数据。
- 第 9 章将介绍一些工具，以处理具有多种相关关系的数据集。
- 第 10 章将介绍正则表达式，即处理字符串的一种强大工具。
- 第 11 章将展示 R 如何保存分类数据。当一个变量的取值范围是有限集合，或者我们不想按字母顺序排列字符串时，就可以使用分类数据。
- 第 12 章将介绍处理日期和日期时间型数据的核心工具。

第 7 章

使用tibble实现简单数据框

7.1 简介

本书使用 tibble 代替传统的 R 数据框。tibble 是一种简单数据框，它对传统数据框的功能进行了一些修改，以便更易于使用。R 是一门古老的语言，其中有些功能在 10 年或 20 年前是适用的，但现在已经过时。在不破坏现有代码的前提下，很难修改 R 的基础功能，因此多数革新都是以扩展包的方式出现的。本章会介绍 tibble 包，其所提供的简单数据框更易于在 tidyverse 中使用。多数情况下，我们会交替使用 tibble 和数据框这两个术语；如果想要特别强调 R 内置的传统数据框，我们会使用 `data.frame` 来表示。

如果读完本章后你还想学习有关 tibble 的更多知识，可以使用 `vignette("tibble")` 命令。

准备工作

我们将在本章中介绍 tidyverse 的核心 R 包之一——tibble 包。

```
library(tidyverse)
```

7.2 创建tibble

本书中使用的所有函数几乎都可以创建 tibble，因为 tibble 是 tidyverse 的标准功能之一。由于多数其他 R 包使用的是标准数据框，因此你可能想要将数据框转换为 tibble。可以使用 `as_tibble()` 函数来完成转换：

```
as_tibble(iris)
#> # A tibble: 150 × 5
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>          <dbl>       <dbl>        <dbl>       <dbl>  <fctr>
```

```
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
#> 4          4.6         3.1          1.5         0.2  setosa
#> 5          5.0         3.6          1.4         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa
#> # ... with 144 more rows
```

可以通过 tibble() 函数使用一个向量来创建新 tibble。tibble() 会自动重复长度为 1 的输入，并可以使用刚刚创建的新变量，如下所示：

```
tibble(
  x = 1:5,
  y = 1,
  z = x ^ 2 + y
)
#> # A tibble: 5 × 3
#>       x     y     z
#>   <int> <dbl> <dbl>
#> 1     1     1     2
#> 2     2     1     5
#> 3     3     1    10
#> 4     4     1    17
#> 5     5     1    26
```

如果你已经非常熟悉 data.frame() 函数，那么请注意 tibble() 函数的功能要少得多：它不能改变输入的类型（例如，不能将字符串转换为因子）、变量的名称，也不能创建行名称。

可以在 tibble 中使用在 R 中无效的变量名称（即**不符合语法的**名称）作为列名称。例如，列名称可以不以字母开头，也可以包含特殊字符（如空格）。要想引用这样的变量，需要使用反引号 ` 将它们括起来：

```
tb <- tibble(
  `:)` = "smile",
  ` ` = "space",
  `2000` = "number"
)
tb
#> # A tibble: 1 × 3
#>    `:)`   ` ` `2000`
#>   <chr> <chr>  <chr>
#> 1 smile space number
```

如果要在 ggplot2 和 dplyr 等其他 R 包中使用这些变量，也需要使用反引号。

创建 tibble 的另一种方法是使用 tribble() 函数，tribble 是 transposed tibble（转置 tibble）的缩写。tribble() 是定制化的，可以对数据按行进行编码：列标题由公式（以 ~ 开头）定义，数据条目以逗号分隔，这样就可以用易读的方式对少量数据进行布局：

```
tribble(
  ~x, ~y, ~z,
  #--|--|----
  "a", 2, 3.6,
  "b", 1, 8.5
```

```
)
#> # A tibble: 2 × 3
#>       x     y     z
#>   <chr> <dbl> <dbl>
#> 1     a     2   3.6
#> 2     b     1   8.5
```

我通常会加一条注释（以 # 开头的行）来明确指出标题行的位置。

7.3　对比tibble与data.frame

tibble 和传统 data.frame 的使用方法主要有两处不同：打印和取子集。

7.3.1　打印

tibble 的打印方法进行了优化，只显示前 10 行结果，并且列也是适合屏幕的，这种方式非常适合大数据集。除了打印列名，tibble 还会打印出列的类型，这项非常棒的功能借鉴于 str() 函数。

```
tibble(
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  e = sample(letters, 1e3, replace = TRUE)
)
#> # A tibble: 1,000 × 5
#>                     a          b     c     d     e
#>                <dttm>     <date> <int> <dbl> <chr>
#> 1 2016-10-10 17:14:14 2016-10-17     1 0.368     h
#> 2 2016-10-11 11:19:24 2016-10-22     2 0.612     n
#> 3 2016-10-11 05:43:03 2016-11-01     3 0.415     l
#> 4 2016-10-10 19:04:20 2016-10-31     4 0.212     x
#> 5 2016-10-10 15:28:37 2016-10-28     5 0.733     a
#> 6 2016-10-11 02:29:34 2016-10-24     6 0.460     v
#> # ... with 994 more rows
```

在打印大数据框时，tibble 的这种设计避免了输出占满整个控制台。但有时需要比默认显示更多的输出，这时就要设置几个选项。

首先，可以明确使用 print() 函数来打印数据框，并控制打印的行数（n）和显示的宽度（width）。width = Inf 可以显示出所有列：

```
nycflights13::flights %>%
  print(n = 10, width = Inf)
```

还可以通过设置以下选项来控制默认的打印方式。

- options(tibble.print_max = n, tibble.pring_min = m)：如果多于 m 行，则只打印出 n 行。options(tibble.print_min = Inf) 表示总是打印所有行。
- options(tibble.width = Inf) 表示总是打印所有列，不考虑屏幕的宽度。

可以使用 package?tibble 命令调出这个包的帮助文件，查看全部的选项列表。

最后一种方式是使用 RStudio 内置的数据查看器，以滚动方式查看整个数据集。进行一长串数据处理操作后，经常会使用这种查看方式：

```
nycflights13::flights %>%
  View()
```

7.3.2 取子集

迄今为止，你学到的所有工具都是作用于整个数据框。如果想要提取单个变量，那么就需要一些新工具，如 $ 和 [[。[[可以按名称或位置提取变量；$ 只能按名称提取变量，但可以减少一些输入：

```
df <- tibble(
  x = runif(5),
  y = rnorm(5)
)

# 按名称提取
df$x
#> [1] 0.434 0.395 0.548 0.762 0.254
df[["x"]]
#> [1] 0.434 0.395 0.548 0.762 0.254

# 按位置提取
df[[1]]
#> [1] 0.434 0.395 0.548 0.762 0.254
```

要想在管道中使用这些提取操作，需要使用特殊的占位符 .：

```
df %>% .$x
#> [1] 0.434 0.395 0.548 0.762 0.254
df %>% .[["x"]]
#> [1] 0.434 0.395 0.548 0.762 0.254
```

与 data.frame 相比，tibble 更严格：它不能进行部分匹配，如果想要访问的列不存在，它会生成一条警告信息。

7.4 与旧代码进行交互

有些比较旧的函数不支持 tibble。如果遇到这种函数，可以使用 as.data.frame() 函数将 tibble 转换回 data.frame：

```
class(as.data.frame(tb))
#> [1] "data.frame"
```

有些旧函数不支持 tibble 的主要原因在于 [的功能。本书没有使用太多的 [，因为 dplyr::filter() 和 dplyr::select() 可以通过更加清晰的代码解决同样的问题（你可以在 16.4.5 节中学到 [的一些使用方法）。对于 R 基础包中的数据框，[有时返回一个数据框，有时返回一个向量。对于 tibble，[则总是返回另一个 tibble。

练习

(1) 如何识别一个对象是否为 tibble？（提示：尝试打印出标准数据框 `mtcars`。）

(2) 对比在 `data.frame` 和等价的 tibble 上进行的以下操作。有何区别？为什么默认的数据框操作会让人感到沮丧？

```
df <- data.frame(abc = 1, xyz = "a")
df$x
df[, "xyz"]
df[, c("abc", "xyz")]
```

(3) 如果将一个变量的名称保存在一个对象中，如 `var <- "mpg"`，如何从 tibble 中提取出这个变量？

(4) 在以下的数据框中练习如何引用不符合语法规则的变量名。

a. 提取名称为 `1` 的变量。
b. 绘制表示变量 `1` 和变量 `2` 关系的散点图。
c. 创建一个名称为 `3` 的新列，其值为列 `2` 除以列 `1`。
d. 将这些列重新命名为 `one`、`two` 和 `three`。

```
annoying <- tibble(
  `1` = 1:10,
  `2` = `1` * 2 + rnorm(length(`1`))
)
```

(5) `tibble::enframe()` 函数的功能是什么？什么时候可以使用这个函数？

(6) 哪个选项控制在 tibble 底部打印的额外列名称的数量？

第 8 章

使用readr进行数据导入

8.1 简介

使用 R 包提供的数据是学习数据科学工具的良好方法，但你总要在某个时间停止学习，开始处理自己的数据。你将在本章中学习如何将纯文本格式的矩形文件读入 R。虽然本章内容只是数据导入的冰山一角，但其中的原则完全适用于其他类型的数据。本章末尾将提供一些有用的 R 包，以处理其他类型的数据。

准备工作

你将在本章中学习如何使用 readr 包将平面文件加载到 R 中，readr 也是 tidyverse 的核心 R 包之一。

```
library(tidyverse)
```

8.2 入门

readr 的多数函数用于将平面文件转换为数据框。

- `read_csv()` 读取逗号分隔文件、`read_csv2()` 读取分号分隔文件（这在用 , 表示小数位的国家非常普遍）、`read_tsv()` 读取制表符分隔文件、`read_delim()` 可以读取使用任意分隔符的文件。
- `read_fwf()` 读取固定宽度的文件。既可以使用 `fwf_widths()` 函数按照宽度来设定域，也可以使用 `fwf_positions()` 函数按照位置来设定域。`read_table()` 读取固定宽度文件的一种常用变体，其中使用空白字符来分隔各列。

- `read_log()` 读取 Apache 风格的日志文件。（但需要检查是否安装了 webreadr 包，https://github.com/Ironholds/webreadr，因为该包位于 `read_log()` 函数的开头，还可以提供很多有用的工具。）

这些函数都具有同样的语法，你完全可以举一反三。在本章余下的内容中，我们将重点介绍 `read_csv()` 函数，不仅因为 CSV 文件是数据存储最常用的形式之一，还因为一旦掌握 `read_csv()` 函数，你就可以将从中学到的知识非常轻松地应用于 readr 的其他函数。

`read_csv()` 函数的第一个参数是最重要的，该参数是要读取的文件的路径：

```
heights <- read_csv("data/heights.csv")
#> Parsed with column specification:
#> cols(
#>   earn = col_double(),
#>   height = col_double(),
#>   sex = col_character(),
#>   ed = col_integer(),
#>   age = col_integer(),
#>   race = col_character()
#> )
```

当运行 `read_csv()` 时，它会打印一份数据列说明，给出每个列的名称和类型。这是 readr 的一项重要功能，8.4 节将继续讨论这项功能。

你还可以提供一个行内 CSV 文件。这种文件非常适合使用 readr 进行实验，以及与他人分享可重现实例的情况。

```
read_csv("a,b,c
1,2,3
4,5,6")
#> # A tibble: 2 × 3
#>       a     b     c
#>   <int> <int> <int>
#> 1     1     2     3
#> 2     4     5     6
```

以上两种情况下，`read_csv()` 函数都使用数据的第一行作为列名称，这是一种常见做法。你或许想在以下两种情况下改变这种做法。

- 有时文件开头会有好几行元数据。你可以使用 `skip = n` 来跳过前 `n` 行；或者使用 `comment = "#"` 来丢弃所有以 `#` 开头的行：

```
read_csv("The first line of metadata
  The second line of metadata
  x,y,z
  1,2,3", skip = 2)
#> # A tibble: 1 × 3
#>       x     y     z
#>   <int> <int> <int>
#> 1     1     2     3

read_csv("# A comment I want to skip
  x,y,z
```

```
  1,2,3", comment = "#")
#> # A tibble: 1 × 3
#>       x     y     z
#>   <int> <int> <int>
#> 1     1     2     3
```

- 数据没有列名称。可以使用 col_names = FALSE 来通知 read_csv() 不要将第一行作为列标题，而是将各列依次标注为 X1 至 Xn：

  ```
  read_csv("1,2,3\n4,5,6", col_names = FALSE)
  #> # A tibble: 2 × 3
  #>      X1    X2    X3
  #>   <int> <int> <int>
  #> 1     1     2     3
  #> 2     4     5     6
  ```

  （"\n" 是非常便捷的快捷方式，用于添加新行。10.2 节介绍了更多关于 "\n" 和其他转义字符的知识。）

  或者你也可以向 col_names 传递一个字符向量，以用作列名称：

  ```
  read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
  #> # A tibble: 2 × 3
  #>       x     y     z
  #>   <int> <int> <int>
  #> 1     1     2     3
  #> 2     4     5     6
  ```

另一个通常需要修改的选项是 na。它设定使用哪个值（或哪些值）来表示文件中的缺失值：

```
read_csv("a,b,c\n1,2,.", na = ".")
#> # A tibble: 1 × 3
#>       a     b     c
#>   <int> <int> <chr>
#> 1     1     2  <NA>
```

如果掌握了以上知识，那么你就可以读取实际中遇到的约 75% 的文件了。你还可以轻松地扩展已经学到的知识，使用 read_tsv() 函数来读取制表符分隔文件，或使用 read_fwf() 函数来读取固定宽度的文件。如果想要读取难度更大的文件，则需要学习更多知识，了解 readr 如何解析每一列，并将其转换为 R 中的向量。

8.2.1 与R基础包进行比较

如果以前使用过 R，那么你肯定很想知道我们为什么不使用 read.csv() 函数。相对于 R 基础包中具有同样功能的函数，我们更喜欢使用 readr 中的函数，理由如下。

- 一般来说，它们比基础模块中的函数速度更快（约快 10 倍）。因为运行时间很长的任务都会有一个进度条，所以你可以看到哪个函数更快。如果只考虑速度的话，还可以尝试使用 data.table::fread()。这个函数与 tidyverse 的兼容性不是很好，但确实更快一些。
- 它们可以生成 tibble，并且不会将字符向量转换为因子，不使用行名称，也不会随意改动列名称。这些都是使用 R 基础包时常见的令人沮丧的事情。

- 它们更易于重复使用。R 基础包中的函数会继承操作系统的功能，并依赖环境变量，因此，可以在你的计算机上正常运行的代码在导入他人计算机时，不一定能正常运行。

8.2.2 练习

(1) 如果一个文件中的域是由“|”分隔的，那么应该使用哪个函数来读取这个文件？

(2) 除了 `file`、`skip` 和 `comment`，还有哪些参数是 `read_csv()` 和 `read_tsv()` 这两个函数共有的？

(3) `read_fwf()` 函数中最重要的参数是什么？

(4) 有时 CSV 文件中的字符串会包含逗号。为了防止引发问题，需要用引号（如 `"` 或 `'`）将逗号围起来。按照惯例，`read_csv()` 默认引号为 `"`，如果想要改变默认值，就要转而使用 `read_delim()` 函数。要想将以下文本读入一个数据框，需要设定哪些参数？

```
"x,y\n1,'a,b'"
```

(5) 找出以下每个行内 CSV 文件中的错误。如果运行代码，会发生什么情况？

```
read_csv("a,b\n1,2,3\n4,5,6")
read_csv("a,b,c\n1,2\n1,2,3,4")
read_csv("a,b\n\"1")
read_csv("a,b\n1,2\na,b")
read_csv("a;b\n1;3")
```

8.3 解析向量

在详细介绍 readr 如何从磁盘读取文件前，我们需要先讨论一下 `parse_*()` 函数族。这些函数接受一个字符向量，并返回一个特定向量，如逻辑、整数或日期向量：

```
str(parse_logical(c("TRUE", "FALSE", "NA")))
#>  logi [1:3] TRUE FALSE NA
str(parse_integer(c("1", "2", "3")))
#>  int [1:3] 1 2 3
str(parse_date(c("2010-01-01", "1979-10-14")))
#>  Date[1:2], format: "2010-01-01" "1979-10-14"
```

这些函数各司其职，且都是 readr 的重要组成部分。一旦掌握了本节中这些单个解析函数的用法，我们就可以继续讨论如何综合使用它们来解析整个文件了。

和 tidyverse 中的所有函数一样，`parse_*()` 函数族的用法是一致的。第一个参数是需要解析的字符向量，`na` 参数设定了哪些字符串应该当作缺失值来处理：

```
parse_integer(c("1", "231", ".", "456"), na = ".")
#> [1]   1 231 NA 456
```

如果解析失败，你会收到一条警告：

```
x <- parse_integer(c("123", "345", "abc", "123.45"))
#> Warning: 2 parsing failures.
#> row col               expected actual
```

```
#>   3  -- an integer                abc
#>   4  -- no trailing characters    .45
```

解析失败的值在输出中是以缺失值的形式存在的：

```
x
#> [1] 123 345  NA  NA
#> attr(,"problems")
#> # A tibble: 2 × 4
#>     row   col               expected actual
#>   <int> <int>                  <chr>  <chr>
#> 1     3    NA             an integer    abc
#> 2     4    NA no trailing characters    .45
```

如果解析失败的值很多，那么就应该使用 `problems()` 函数来获取完整的失败信息集合。这个函数会返回一个 tibble，你可以使用 dplyr 包来进行处理：

```
problems(x)
#> # A tibble: 2 × 4
#>     row   col               expected actual
#>   <int> <int>                  <chr>  <chr>
#> 1     3    NA             an integer    abc
#> 2     4    NA no trailing characters    .45
```

在解析函数的使用方面，最重要的是要知道有哪些解析函数，以及每种解析函数用来处理哪种类型的输入。具体来说，重要的解析函数有 8 种。

- `parse_logical()` 和 `parse_integer()` 函数分别解析逻辑值和整数。因为这两个解析函数基本不会出现问题，所以我们不再进行更多介绍。
- `parse_double()` 是严格的数值型解析函数，`parse_number()` 则是灵活的数值型解析函数。这两个函数要比你预想的更复杂，因为世界各地书写数值的方式不尽相同。
- `parse_character()` 函数似乎太过简单，甚至没必要存在。但一个棘手的问题使得这个函数变得非常重要：字符编码。
- `parse_factor()` 函数可以创建因子，R 使用这种数据结构来表示分类变量，该变量具有固定数目的已知值。
- `parse_datetime()`、`parse_date()` 和 `parse_time()` 函数可以解析不同类型的日期和时间。它们是最复杂的，因为有太多不同的日期书写形式。

我们将在以下各节中更加详细地介绍这些解析函数。

8.3.1　数值

解析数值似乎是非常直截了当的，但以下 3 个问题增加了数值解析的复杂性。

- 世界各地的人们书写数值的方式不尽相同。例如，有些国家使用 `.` 来分隔实数中的整数和小数部分，而有些国家则使用 `,`。
- 数值周围经常有表示某种意义的其他字符，如 $1000 或 10%。
- 数值经常包含“分组”，以便更易读，如 1 000 000，而且世界各地用来分组的字符也不尽相同。

为了解决第一个问题，readr 使用了“地区”这一概念，这是可以按照不同地区设置解析选项的一个对象。在解析数值时，最重要的选项就是用来表示小数点的字符。通过创建一个新的地区对象并设定 decimal_mark 参数，可以覆盖 . 的默认值：

```
parse_double("1.23")
#> [1] 1.23
parse_double("1,23", locale = locale(decimal_mark = ","))
#> [1] 1.23
```

readr 的默认地区是 US-centric，因为 R 是以美国为中心的（也就是说，R 基础包的文档是用美式英语写成的）。获取默认地区设置的另一种方法是利用操作系统，但这种方法很难奏效，更重要的是，这会让你的代码很脆弱：即使可以在你的计算机上良好运行，但通过电子邮件分享给另一个国家的同事时，就可能失效。

parse_number() 解决了第二个问题：它可以忽略数值前后的非数值型字符。这个函数特别适合处理货币和百分比，也可以提取嵌在文本中的数值：

```
parse_number("$100")
#> [1] 100
parse_number("20%")
#> [1] 20
parse_number("It cost $123.45")
#> [1] 123
```

组合使用 parse_number() 和地区设置可以解决最后一个问题，因为 parse_number() 可以忽略“分组符号”：

```
# 适用于美国
parse_number("$123,456,789")
#> [1] 1.23e+08

# 适用于多数欧洲国家
parse_number(
  "123.456.789",
  locale = locale(grouping_mark = ".")
)
#> [1] 1.23e+08

# 适用于瑞士
parse_number(
  "123'456'789",
  locale = locale(grouping_mark = "'")
)
#> [1] 1.23e+08
```

8.3.2　字符串

parse_character() 函数似乎真的很简单，只要返回输入值就可以了。问题是生活没这么简单，因为同一个字符串有多种表示方式。为了理解个中缘由，我们需要深入介绍一下计算机是如何表示字符串的。在 R 中，我们可以使用 charToRaw() 函数获得一个字符串的底层表示：

```
charToRaw("Hadley")
#> [1] 48 61 64 6c 65 79
```

每个十六进制数表示信息的一个字节：48 是 H、61 是 a 等。从十六进制数到字符的这种映射称为编码，这个示例中的编码方式称为 ASCII。ASCII 可以非常好地表示英文字符，因为它就是美国信息交换标准代码（American Standard Code for Information Interchange）的缩写。

对于英语之外的其他语言，事情就变得更加复杂了。计算机发展的早期阶段有很多为非英语字符进行编码的标准，它们之间甚至是相互矛盾的。要想正确表示一个字符串，不仅需要知道它的值，还要知道其编码方式。例如，Latin1（即 ISO-8859-1，用于西欧语言）和 Latin2（即 ISO-8859-2，用于东欧语言）是两种常用的编码方式。字节 b1 在 Latin1 中表示“±”，但在 Latin2 中则表示“ą”！好在现在有一种几乎所有语言都支持的标准：UTF-8。UTF-8 可以为现在人类使用的所有字符进行编码，同时还支持很多特殊字符（如表情符号！）。

readr 全面支持 UTF-8：当读取数据时，它假设数据是 UTF-8 编码的，并总是使用 UTF-8 编码写入数据。这是非常好的默认方式，但对于从不支持 UTF-8 的那些旧系统中产生的数据则无能为力。遇到这种情况时，你的字符串打印出来就是一堆乱码。有时只有一两个字符是乱码；有时则完全不知所云。例如：

```
x1 <- "El Ni\xf1o was particularly bad this year"
x2 <- "\x82\xb1\x82\xf1\x82\xc9\x82\xbf\x82\xcd"
```

要想解决这个问题，需要在 parse_character() 函数中设定编码方式：

```
parse_character(x1, locale = locale(encoding = "Latin1"))
#> [1] "El Niño was particularly bad this year"
parse_character(x2, locale = locale(encoding = "Shift-JIS"))
#> [1] "こんにちは"
```

如何才能找到正确的编码方式呢？如果足够幸运，那么编码方式可能就写在数据文档中。遗憾的是这种情况非常罕见，因此 readr 提供了 guess_encoding() 函数来帮助你找出编码方式。但这个函数并非万无一失，如果有大量文本（不像本例），效果就会更好，它确实是一个良好的起点。希望试验几次后，你就能够找到正确的编码方式：

```
guess_encoding(charToRaw(x1))
#>     encoding confidence
#> 1 ISO-8859-1       0.46
#> 2 ISO-8859-9       0.23
guess_encoding(charToRaw(x2))
#>   encoding confidence
#> 1   KOI8-R       0.42
```

guess_encoding() 的第一个参数可以是一个文件路径，也可以是一个原始向量（适用于字符串已经在 R 中的情况），就像本示例一样。

编码问题博大精深，这里我们只是蜻蜓点水式地介绍一下。如果想要学习更多相关知识，我们推荐你阅读 http://kunststube.net/encoding/ 中的详细说明。

8.3.3 因子

R 使用因子表示取值范围是已知集合的分类变量。如果 `parse_factor()` 函数的 `levels` 参数被赋予一个已知向量，那么只要存在向量中没有的值，就会生成一条警告：

```
fruit <- c("apple", "banana")
parse_factor(c("apple", "banana", "bananana"), levels = fruit)
#> Warning: 1 parsing failure.
#> row col           expected   actual
#>   3  -- value in level set bananana
#> [1] apple  banana <NA>
#> attr(,"problems")
#> # A tibble: 1 × 4
#>     row   col          expected   actual
#>   <int> <int>             <chr>    <chr>
#> 1     3    NA value in level set bananana
#> Levels: apple banana
```

如果有很多问题条目的话，通常更简单的做法是将它们作为字符向量，然后使用将在第 10 章和第 11 章中介绍的工具来进行数据清理。

8.3.4 日期、日期时间与时间

根据需要的是日期型数据（从 1970-01-01 开始的天数）、日期时间型数据（从 1970-01-01 午夜开始的秒数），或者是时间型数据（从午夜开始的秒数），我们可以在 3 种解析函数之间进行选择。在没有使用任何附加参数时调用，具体情况如下。

- `parse_datetime()` 期待的是符合 ISO 8601 标准的日期时间。ISO 8601 是一种国际标准，其中日期的各个部分按从大到小的顺序排列，即年、月、日、小时、分钟、秒：

```
parse_datetime("2010-10-01T2010")
#> [1] "2010-10-01 20:10:00 UTC"

# 如果时间被省略了，那么它就会被设置为午夜
parse_datetime("20101010")
#> [1] "2010-10-10 UTC"
```

  这是最重要的日期 / 时间标准，如果经常使用日期和时间，我们建议你阅读一下维基百科上的 ISO 8601 标准。

- `parse_date()` 期待的是四位数的年份、一个 `-` 或 `/`、月、一个 `-` 或 `/`，然后是日：

```
parse_date("2010-10-01")
#> [1] "2010-10-01"
```

- `parse_time()` 期待的是小时、`:`、分钟、可选的 `:` 和秒，以及一个可选的 a.m./p.m. 标识符：

```
library(hms)
parse_time("01:10 am")
#> 01:10:00
parse_time("20:10:01")
#> 20:10:01
```

  因为 R 基础包中没有能够很好表示时间数据的内置类，所以我们使用 hms 包提供的时间类。

如果这些默认设置不适合你的数据，那么你可以提供自己的日期时间格式，格式由以下各部分组成。

年

`%Y`（4 位数）。

`%y`（2 位数；00-69 → 2000-2069、70-99 → 1970-1999）。

月

`%m`（2 位数）。

`%b`（简写名称，如 Jan）。

`%B`（完整名称，如 January）。

日

`%d`（1 位或 2 位数）。

`%e`（2 位数）

时间

`%H`（0-23 小时）。

`%I`（0-12 小时，必须和 `%p` 一起使用）。

`%p`（表示 a.m./p.m.）。

`%M`（分钟）。

`%S`（整数秒）。

`%OS`（实数秒）。

`%Z`（时区，`America/Chicage` 这样的名称）。注意，要当心缩写。如果你是美国人，注意 EST 是加拿大没有夏时制的一个时区。它表示东部标准时间！我们还会在 12.5 节中继续讨论这个话题。

`%z`（与国际标准时间的时差，如 `+0800`）。

非数值字符

`%.`（跳过一个非数值字符）。

`%*`（跳过所有非数值字符）。

找出正确格式的最好方法是创建几个解析字符向量的示例，并使用某种解析函数进行测试。例如：

```
parse_date("01/02/15", "%m/%d/%y")
#> [1] "2015-01-02"
parse_date("01/02/15", "%d/%m/%y")
#> [1] "2015-02-01"
parse_date("01/02/15", "%y/%m/%d")
#> [1] "2001-02-15"
```

如果对非英语月份名称使用 %b 或 %B，那么你就需要在 locale() 函数中设置 lang 参数。查看 date_names_langs() 函数中的内置语言列表，如果你的语言没有包括在内，那么可以使用 date_names() 函数创建自己的月份和日期名称：

```
parse_date("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
#> [1] "2015-01-01"
```

8.3.5 练习

(1) locale() 函数中最重要的参数是什么？

(2) 如果将 decimal_mark 和 grouping_mark 设为同一个字符，会发生什么情况？如果将 decimal_mark 设为 ,，grouping_mark 的默认值会发生什么变化？如果将 grouping_mark 设为 .，decimal_mark 的默认值会发生什么变化？

(3) 我们没有讨论过 locale() 函数的 date_format 和 time_format 选项，它们的作用是什么？构建一个示例，说明它们在何种情况下是有用的？

(4) 如果你不是居住在美国，创建一个新的地区对象，并封装你最常读取的文件类型的相关设置。

(5) read_csv() 和 read_csv2() 之间的区别是什么？

(6) 欧洲最常用的编码方式是什么？亚洲最常用的编码方式是什么？可以使用 google 找出答案。

(7) 生成正确形式的字符串来解析以下日期和时间。

```
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14" # 2014年12月30日
t1 <- "1705"
t2 <- "11:15:10.12 PM"
```

8.4 解析文件

现在你已经学会了如何解析单个向量，接下来我们就回到本章的最初目标，研究 readr 是如何解析文件的。你将在本节中学到以下两种新技能。

- readr 如何自动猜出文件每列的数据类型。
- 如何修改默认设置。

8.4.1 策略

readr 使用一种启发式过程来确定每列的类型：先读取文件的前 1000 行，然后使用（相对保守的）某种启发式算法确定每列的类型。可以使用字符向量模拟这个过程，先使用 guess_parser() 函数返回 readr 最可信的猜测，接着 parse_guess() 函数使用这个猜测来解析列：

```
guess_parser("2010-10-01")
#> [1] "date"
guess_parser("15:01")
#> [1] "time"
guess_parser(c("TRUE", "FALSE"))
#> [1] "logical"
guess_parser(c("1", "5", "9"))
#> [1] "integer"
guess_parser(c("12,352,561"))
#> [1] "number"

str(parse_guess("2010-10-10"))
#>  Date[1:1], format: "2010-10-10"
```

这个启发式过程会尝试以下每种数据类型，直至找到匹配的类型。

逻辑值

只包括 F、T、FALSE 和 TRUE。

整数

只包括数值型字符（以及 -）。

双精度浮点数

只包括有效的双精度浮点数（也包括 4.5e-5 这样的数值）。

数值

只包括带有分组符号的有效双精度浮点数。

时间

与默认的 `time_format` 匹配的值。

日期

与默认的 `date_format` 匹配的值。

日期时间

符合 ISO 8601 标准的任何日期。

如果以上类型均不匹配，那么这一列就还是一个字符串向量。

8.4.2 问题

这些默认设置对更大的文件并不是一直有效的。以下是两个主要问题。

- 前 1000 行可能是一种特殊情况，readr 猜测出的类型不足以代表整个文件。例如，一列双精度数值的前 1000 行可能都是整数。
- 列中可能含有大量缺失值。如果前 1000 行中都是 `NA`，那么 readr 会猜测这是一个字符向量，但你其实想将这一列解析为更具体的值。

readr 中包含了一份非常有挑战性的 CSV 文件，该文件可以说明以上两个问题。

```
challenge <- read_csv(readr_example("challenge.csv"))
#> Parsed with column specification:
#> cols(
#>   x = col_integer(),
#>   y = col_character()
#> )
#> Warning: 1000 parsing failures.
#>  row col               expected             actual
#> 1001   x no trailing characters .23837975086644292
#> 1002   x no trailing characters .41167997173033655
#> 1003   x no trailing characters .7460716762579978
#> 1004   x no trailing characters .723450553836301
#> 1005   x no trailing characters .614524137461558
#> .... ... ..................... ..................
#> See problems(...) for more details.
```

（注意 readr_example() 函数的用法，它可以找到包含在 R 包中的文件的路径。）

以上的输出可以分为两部分：从前 1000 行中猜测出的列类型与前 5 条解析失败记录。我们总是应该使用 problems() 函数明确地列出这些失败记录，以便更加深入地探究其中的问题：

```
problems(challenge)
#> # A tibble: 1,000 × 4
#>     row   col               expected             actual
#>   <int> <chr>                  <chr>              <chr>
#> 1  1001     x no trailing characters .23837975086644292
#> 2  1002     x no trailing characters .41167997173033655
#> 3  1003     x no trailing characters  .7460716762579978
#> 4  1004     x no trailing characters   .723450553836301
#> 5  1005     x no trailing characters   .614524137461558
#> 6  1006     x no trailing characters   .473980569280684
#> # ... with 994 more rows
```

一列列地处理，直至解决所有问题，是一种良好策略。这里我们可以看到 x 列中存在大量解析问题——整数后面有拖尾字符。这说明我们应该使用双精度解析函数。

为了解决这个问题，首先，复制列类型并将其粘贴到初始调用中：

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_integer(),
    y = col_character()
  )
)
```

接着修改 x 列的类型：

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_character()
  )
)
```

这样就解决了第一个问题，但如果查看最后几行的话，你会发现保存在字符向量中的其实是日期数据：

```
tail(challenge)
#> # A tibble: 6 × 2
#>       x          y
#>   <dbl>      <chr>
#> 1 0.805 2019-11-21
#> 2 0.164 2018-03-29
#> 3 0.472 2014-08-04
#> 4 0.718 2015-08-16
#> 5 0.270 2020-02-04
#> 6 0.608 2019-01-06
```

设定 y 为日期列可以解决这个问题：

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_date()
  )
)
tail(challenge)
#> # A tibble: 6 × 2
#>       x          y
#>   <dbl>     <date>
#> 1 0.805 2019-11-21
#> 2 0.164 2018-03-29
#> 3 0.472 2014-08-04
#> 4 0.718 2015-08-16
#> 5 0.270 2020-02-04
#> 6 0.608 2019-01-06
```

每个 `parse_xyz()` 函数都有一个对应的 `col_xyz()` 函数。如果数据已经保存在 R 的字符向量中，那么你可以使用 `parse_xyz()`；如果想要告诉 readr 如何加载数据，则应该使用 `col_xyz()`。

我们强烈建议你总是提供 `col_types` 参数，从 readr 打印出的输出中可以知道它的值。这可以确保数据导入脚本的一致性，并可以重复使用。如果不提供这个参数，而是依赖猜测的类型，那么当数据发生变化时，readr 会继续读入数据。如果想要严格解析，可以使用 `stop_for_problems()` 函数：当出现任何解析问题时，它会抛出一个错误，并终止脚本。

8.4.3 其他策略

我们再介绍其他几种有助于解析文件的通用策略。

- 在前面的示例中，我们的运气太差了：如果比默认方式再多检查 1 行，我们就能一蹴而就，解析成功。

```
challenge2 <- read_csv(
                readr_example("challenge.csv"),
                guess_max = 1001
              )
```

```
#> Parsed with column specification:
#> cols(
#>   x = col_double(),
#>   y = col_date(format = "")
#> )
challenge2
#> # A tibble: 2,000 × 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
```

- 有时如果将所有列都作为字符向量读入的话，会更容易诊断出问题：

  ```
  challenge2 <- read_csv(readr_example("challenge.csv"),
    col_types = cols(.default = col_character())
  )
  ```

  这种方式结合 type_convert() 函数使用时特别有效，后者可以在数据框的字符列上应用启发式解析过程：

  ```
  df <- tribble(
    ~x,  ~y,
    "1", "1.21",
    "2", "2.32",
    "3", "4.56"
  )
  df
  #> # A tibble: 3 × 2
  #>       x     y
  #>   <chr> <chr>
  #> 1     1  1.21
  #> 2     2  2.32
  #> 3     3  4.56

  # 注意列类型
  type_convert(df)
  #> Parsed with column specification:
  #> cols(
  #>   x = col_integer(),
  #>   y = col_double()
  #> )
  #> # A tibble: 3 × 2
  #>       x     y
  #>   <int> <dbl>
  #> 1     1  1.21
  #> 2     2  2.32
  #> 3     3  4.56
  ```

- 如果正在读取一个非常大的文件，那么你应该将 n_max 设置为一个较小的数，比如 10 000 或者 100 000。这可以让你在解决常见问题时加快重复试验的过程。

- 如果遇到严重的解析问题，有时使用 read_lines() 函数按行读入字符向量会更容易，甚至可以使用 read_file() 函数读入一个长度为 1 的字符向量。接着你可以使用后面将学到的字符串解析技能来解析各种各样的数据形式。

8.5 写入文件

readr 还提供了两个非常有用的函数，用于将数据写回到磁盘：write_csv() 和 write_tsv()。这两个函数输出的文件能够顺利读取的概率更高，因为：

- 它们总是使用 UTF-8 对字符串进行编码；
- 它们使用 ISO 8601 格式来保存日期和日期时间数据，以便这些数据不论在何种环境下都更容易解析。

如果想要将 CSV 文件导为 Excel 文件，可以使用 write_excel_csv() 函数，该函数会在文件开头写入一个特殊字符（字节顺序标记），告诉 Excel 这个文件使用的是 UTF-8 编码。

这几个函数中最重要的参数是 x（要保存的数据框）和 path（保存文件的位置）。还可以使用 na 参数设定如何写入缺失值，如果想要追加到现有的文件，需要设置 append 参数：

```
write_csv(challenge, "challenge.csv")
```

注意，当保存为 CSV 文件时，类型信息就丢失了：

```
challenge
#> # A tibble: 2,000 × 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
write_csv(challenge, "challenge-2.csv")
read_csv("challenge-2.csv")
#> Parsed with column specification:
#> cols(
#>   x = col_double(),
#>   y = col_character()
#> )
#> # A tibble: 2,000 × 2
#>       x     y
#>   <dbl> <chr>
#> 1   404  <NA>
#> 2  4172  <NA>
#> 3  3004  <NA>
#> 4   787  <NA>
#> 5    37  <NA>
#> 6  2332  <NA>
#> # ... with 1,994 more rows
```

这使得 CSV 文件在暂存临时结果时有些不可靠——每次加载时都要重建列类型。以下是

两种替代方式。

- `write_rds()` 和 `read_rds()` 函数是对基础函数 `readRDS()` 和 `saveRDS()` 的统一包装。前者可以将数据保存为 R 自定义的二进制格式，称为 RDS 格式：

```
write_rds(challenge, "challenge.rds")
read_rds("challenge.rds")
#> # A tibble: 2,000 × 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
```

- feather 包实现了一种快速二进制格式，可以在多个编程语言间共享：

```
library(feather)
write_feather(challenge, "challenge.feather")
read_feather("challenge.feather")
#> # A tibble: 2,000 x 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
```

feather 要比 RDS 速度更快，而且可以在 R 之外使用。RDS 支持列表列（我们将在第 19 章中介绍），feather 目前还不行。

8.6　其他类型的数据

要想将其他类型的数据导入 R 中，我们建议首先从下列的 tidyverse 包开始。它们当然远非完美，但确实是一个很好的起点。对矩形数据来说：

- haven 可以读取 SPSS、Stata 和 SAS 文件；
- readxl 可以读取 Excel 文件（.xls 和 .xlsx 均可）；
- 配合专用的数据库后端程序（如 RMySQL、RSQLite、RPostgreSQL 等），DBI 可以对相应数据库进行 SQL 查询，并返回一个数据框。

对于层次数据，可以使用 jsonlite（由 JeroenOoms 开发）读取 JSON 串，使用 xml2 读取 XML 文件。Jenny Bryan 在 https://jennybc.github.io/purrr-tutorial/ 中提供了一些非常好的示例。

对于其他的文件类型，可以学习一下 R 数据导入 / 导出手册（https://cran.r-project.org/doc/manuals/r-release/R-data.html），以及 rio 包（https://github.com/leeper/rio）。

第 9 章

使用dplyr处理关系数据

9.1　简介

只涉及一张数据表的数据分析是非常罕见的。通常来说，你会有很多个数据表，而且必须综合使用它们才能回答你所感兴趣的问题。存在于多个表中的这种数据统称为**关系数据**，因为重要的是数据间的关系，而不是单个数据集。

关系总是定义于两张表之间。其他所有关系都是建立在这种简单思想之上：三张或更多表之间的关系总是可以用每两个表之间关系表示出来。有时关系涉及的两个表甚至就是同一张！例如，如果你有一张人员表，那么其中某个人与其父母的关系就是这种情况。

要想处理关系数据，你需要能够在两张表之间进行的操作。我们设计了三类操作来处理关系数据。

- **合并连接**：向数据框中加入新变量，新变量的值是另一个数据框中的匹配观测。
- **筛选连接**：根据是否匹配另一个数据框中的观测，筛选数据框中的观测。
- **集合操作**：将观测作为集合元素来处理。

关系数据最常见于**关系数据库管理系统**（relational database management system，RDBMS），该系统几乎囊括了所有的现代数据库。如果之前使用过数据库，那你肯定使用过 SQL。如果是这样的话，你会发现本章中的很多概念都似曾相识，尽管其在 dplyr 中的表达形式略微不同。一般来说，dplyr 要比 SQL 更容易使用，因为前者是专门用于进行数据分析的。在进行常用的数据分析操作时，dplyr 非常得心应手，反之，它并不擅长数据分析中不常用的那些操作。

准备工作

我们使用 dplyr 的一些函数来研究一下 nycflights13 中的关系数据，这些函数可以在两张数

据表间进行操作。

```
library(tidyverse)
library(nycflights13)
```

9.2 nycflights13

我们将使用 nycflights13 包来学习关系数据。nycflights13 中包含了与 `flights` 相关的 4 个 tibble，我们已经在第 3 章中使用过 `flights` 表了。

- `airlines`：可以根据航空公司的缩写码查到公司全名。

```
airlines
#> # A tibble: 16 × 2
#>   carrier                   name
#>     <chr>                  <chr>
#> 1      9E       Endeavor Air Inc.
#> 2      AA  American Airlines Inc.
#> 3      AS    Alaska Airlines Inc.
#> 4      B6         JetBlue Airways
#> 5      DL    Delta Air Lines Inc.
#> 6      EV ExpressJet Airlines Inc.
#> # ... with 10 more rows
```

- `airports`：给出了每个机场的信息，通过 `faa` 机场编码进行标识。

```
airports
#> # A tibble: 1,396 × 7
#>     faa                           name   lat   lon
#>   <chr>                          <chr> <dbl> <dbl>
#> 1   04G              Lansdowne Airport 41.1 -80.6
#> 2   06A  Moton Field Municipal Airport 32.5 -85.7
#> 3   06C            Schaumburg Regional 42.0 -88.1
#> 4   06N                Randall Airport 41.4 -74.4
#> 5   09J          Jekyll Island Airport 31.1 -81.4
#> 6   0A9 Elizabethton Municipal Airport  36.4 -82.2
#> # ... with 1,390 more rows, and 3 more variables:
#> #   alt <int>, tz <dbl>, dst <chr>
```

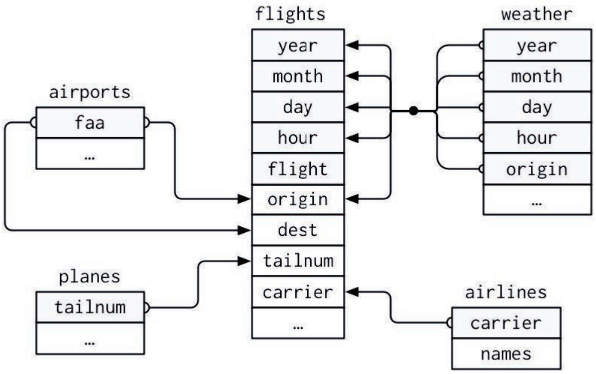
- `planes`：给出了每架飞机的信息，通过 `tailnum` 进行标识。

```
planes
#> # A tibble: 3,322 × 9
#>   tailnum  year                    type
#>     <chr> <int>                   <chr>
#> 1  N10156  2004 Fixed wing multi engine
#> 2  N102UW  1998 Fixed wing multi engine
#> 3  N103US  1999 Fixed wing multi engine
#> 4  N104UW  1999 Fixed wing multi engine
#> 5  N10575  2002 Fixed wing multi engine
#> 6  N105UW  1999 Fixed wing multi engine
#> # ... with 3,316 more rows, and 6 more variables:
#> #  manufacturer <chr>, model <chr>, engines <int>,
#> #  seats <int>, speed <int>, engine <chr>
```

- `weather`：给出了纽约机场每小时的天气状况。

```
weather
#> # A tibble: 26,130 × 15
#>   origin  year month   day  hour  temp  dewp humid
#>    <chr> <dbl> <dbl> <int> <int> <dbl> <dbl> <dbl>
#> 1    EWR  2013     1     1     0  37.0  21.9  54.0
#> 2    EWR  2013     1     1     1  37.0  21.9  54.0
#> 3    EWR  2013     1     1     2  37.9  21.9  52.1
#> 4    EWR  2013     1     1     3  37.9  23.0  54.5
#> 5    EWR  2013     1     1     4  37.9  24.1  57.0
#> 6    EWR  2013     1     1     6  39.0  26.1  59.4
#> # ... with 2.612e+04 more rows, and 7 more variables:
#> #   wind_dir <dbl>, wind_speed <dbl>, wind_gust <dbl>,
#> #   precip <dbl>, pressure <dbl>, visib <dbl>,
#> #   time_hour <dttm>
```

展示不同数据表之间关系的一种方法是绘制图形。



这个图有点让人眼花缭乱，但跟实际工作中的一些图比起来，可以说是相当简洁了。理解这种图的关键是，记住每种关系只与两张表有关。不需要弄清楚所有的事情，只要明白你所关心的表格间的关系即可。

对于 nycflights13 包中的表来说：

- `flights` 与 `planes` 通过单变量 `tailnum` 相连；
- `flights` 与 `airlines` 通过变量 `carrier` 相连；
- `flights` 与 `airports` 通过两种方式相连（变量 `origin` 和 `dest`）；
- `flights` 与 `weather` 通过变量 `origin`（位置）以及 `year`、`month`、`day` 和 `hour`（时间）相连。

练习

(1) 假设想要画出每架飞机从起点到终点的（近似）飞行路线，需要哪些变量？需要组合哪些表格？

(2) 我们忘记画出 `weather` 和 `airports` 之间的关系了，它们之间的关系是什么？如何在图中表示？

(3) weather 表中仅包含起点机场（纽约）的信息。如果它包含美国所有机场的天气记录，那么应如何定义其与 flights 之间的关系？

(4) 我们知道每年的有些日子是“特殊的”，这些日子中乘飞机的人比平时要少。如何将这种数据表示为一个数据框？这个表的主键是什么？它与现有表格之间的关系是怎样的？

9.3 键

用于连接每对数据表的变量称为**键**。键是能唯一标识观测的变量（或变量集合）。简单情况下，单个变量就足以标识一个观测。例如，每架飞机都可以由 tailnum 唯一标识。其他情况可能需要多个变量。例如，要想标识 weather 中的观测，你需要 5 个变量：year、month、day、hour 和 origin。

键的类型有两种。

- **主键**：唯一标识其所在数据表中的观测。例如，planes$tailnum 是一个主键，因为其可以唯一标识 planes 表中的每架飞机。
- **外键**：唯一标识另一个数据表中的观测。例如，flights$tailnum 是一个外键，因为其出现在 flights 表中，并可以将每次航班与唯一一架飞机匹配。

一个变量既可以是主键，也可以是外键。例如，origin 是 weather 表主键的一部分，同时也是 airports 表的外键。

一旦识别出表的主键，最好验证一下，看看它们能否真正唯一标识每个观测。一种验证方法是对主键进行 count() 操作，然后查看是否有 n 大于 1 的记录：

```
planes %>%
  count(tailnum) %>%
  filter(n > 1)
#> # A tibble: 0 × 2
#> # ... with 2 variables: tailnum <chr>, n <int>
weather %>%
  count(year, month, day, hour, origin) %>%
  filter(n > 1)
#> Source: local data frame [0 x 6]
#> Groups: year, month, day, hour [0]
#>
#> # ... with 6 variables: year <dbl>, month <dbl>, day <int>,
#> #   hour <int>, origin <chr>, n <int>
```

有时数据表没有明确的主键：每行都是一个观测，但没有一个变量组合能够明确地标识它。例如，flights 表中的主键是什么？你可能认为是日期加航班号或者是日期加机尾编号，但这两种组合都不是唯一标识：

```
flights %>%
  count(year, month, day, flight) %>%
  filter(n > 1)
#> Source: local data frame [29,768 x 5]
#> Groups: year, month, day [365]
#>
```

```
#>    year month   day flight     n
#>   <int> <int> <int>  <int> <int>
#> 1  2013     1     1      1     2
#> 2  2013     1     1      3     2
#> 3  2013     1     1      4     2
#> 4  2013     1     1     11     3
#> 5  2013     1     1     15     2
#> 6  2013     1     1     21     2
#> # ... with 2.976e+04 more rows

flights %>%
  count(year, month, day, tailnum) %>%
  filter(n > 1)
#> Source: local data frame [64,928 x 5]
#> Groups: year, month, day [365]
#>
#>    year month   day tailnum     n
#>   <int> <int> <int>   <chr> <int>
#> 1  2013     1     1  N0EGMQ     2
#> 2  2013     1     1  N11189     2
#> 3  2013     1     1  N11536     2
#> 4  2013     1     1  N11544     3
#> 5  2013     1     1  N11551     2
#> 6  2013     1     1  N12540     2
#> # ... with 6.492e+04 more rows
```

当开始处理这份数据时，我们天真地假设了每个航班号每天只用一次，因为这样就非常容易与某个特定航班来交流问题。但很遗憾，真实情况并不是这样的。如果一张表没有主键，有时就需要使用 `mutate()` 函数和 `row_number()` 函数为表加上一个主键。这样一来，如果你完成了一些筛选工作，并想要使用原始数据检查的话，就可以更容易地匹配观测。这种主键称为**代理键**。

主键与另一张表中与之对应的外键可以构成**关系**。关系通常是一对多的。例如，每个航班只有一架飞机，但每架飞机可以飞多个航班。在另一些数据中，你有时还会遇到一对一的关系。你可以将这种关系看作一对多关系的特殊情况。你可以使用多对一关系加上一对多关系来构造多对多关系。例如，在这份数据中，航空公司与机场之间存在着多对多关系：每个航空公司可以使用多个机场，每个机场可以服务多个航空公司。

练习

(1) 向 `flights` 添加一个代理键。

(2) 找出以下各数据集中的键。

a. `Lahman::Batting`

b. `babynames::babynames`

c. `nasaweather::atmos`

d. `fueleconomy::vehicles`

e. `ggplot2::diamonds`

（你可能需要安装一些 R 包，并阅读一些文档。）

(3) 画图说明 Lahman 包中的 `Batting`、`Master` 和 `Salaries` 表之间的关系。画另一张图来说明 `Master`、`Managers` 和 `AwardsManagers` 表之间的关系。

应该如何描绘 `Batting`、`Pitching` 和 `Fielding` 表之间的关系？

9.4 合并连接

本节将介绍用于组合两个表格的第一种工具，即**合并连接**。合并连接可以将两个表格中的变量组合起来，它先通过两个表格的键匹配观测，然后将一个表格中的变量复制到另一个表格中。

和 `mutate()` 函数一样，连接函数也会将变量添加在表格的右侧，因此如果表格中已经有了很多变量，那么新变量就不会显示出来。为了解决这个问题，我们建立一个简化的数据集，以便更易看到示例数据集中发生的变化：

```
flights2 <- flights %>%
  select(year:day, hour, origin, dest, tailnum, carrier)
flights2
#> # A tibble: 336,776 × 8
#>    year month   day  hour origin  dest tailnum carrier
#>   <int> <int> <int> <dbl>  <chr> <chr>   <chr>   <chr>
#> 1  2013     1     1     5    EWR   IAH  N14228      UA
#> 2  2013     1     1     5    LGA   IAH  N24211      UA
#> 3  2013     1     1     5    JFK   MIA  N619AA      AA
#> 4  2013     1     1     5    JFK   BQN  N804JB      B6
#> 5  2013     1     1     6    LGA   ATL  N668DN      DL
#> 6  2013     1     1     5    EWR   ORD  N39463      UA
#> # ... with 3.368e+05 more rows
```

（记住，如果使用的是 RStudio，你还可以使用 `View()` 来解决这个问题。）

假设想要将航空公司的全名加入 `flights2` 数据集，你可以通过 `left_join()` 函数组合 `airlines` 和 `flights2` 数据框：

```
flights2 %>%
  select(-origin, -dest) %>%
  left_join(airlines, by = "carrier")
#> # A tibble: 336,776 × 7
#>    year month   day  hour tailnum carrier
#>   <int> <int> <int> <dbl>   <chr>   <chr>
#> 1  2013     1     1     5  N14228      UA
#> 2  2013     1     1     5  N24211      UA
#> 3  2013     1     1     5  N619AA      AA
#> 4  2013     1     1     5  N804JB      B6
#> 5  2013     1     1     6  N668DN      DL
#> 6  2013     1     1     5  N39463      UA
#> # ... with 3.368e+05 more rows, and 1 more variable:
#> #   name <chr>
```

将航空公司数据连接到 `flights2` 的结果产生了一个新变量：`name`。这就是我们将这种连接称为合并连接的原因。对于这个示例，我们可以通过 `mutate()` 函数和 R 的取子集操作达到同样的效果：

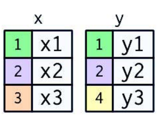
```
flights2 %>%
  select(-origin, -dest) %>%
  mutate(name = airlines$name[match(carrier, airlines$carrier)])
#> # A tibble: 336,776 × 7
#>    year month   day  hour tailnum carrier
#>   <int> <int> <int> <dbl>   <chr>   <chr>
#> 1  2013     1     1     5  N14228      UA
#> 2  2013     1     1     5  N24211      UA
#> 3  2013     1     1     5  N619AA      AA
#> 4  2013     1     1     5  N804JB      B6
#> 5  2013     1     1     6  N668DN      DL
#> 6  2013     1     1     5  N39463      UA
#> # ... with 3.368e+05 more rows, and 1 more variable:
#> #   name <chr>
```

但这种方式很难推广到需要匹配多个变量的情况，而且需要仔细阅读代码才能搞清楚操作目的。

下一节将详细阐释合并连接的工作原理。首先，我们将介绍连接的一种可视化表示。接着使用这种可视化表示来解释 4 种合并连接：1 种内连接和 3 种外连接。在处理实际数据时，键并不能总是唯一地标识观测，因此我们接下来将讨论如何处理不能唯一匹配的情况。最后，我们将介绍如何通知 dplyr 哪个变量是给定连接的键。

9.4.1　理解连接

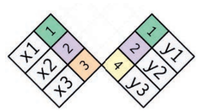
为了帮助你掌握连接的工作原理，我们将介绍用图形来表示连接的一种方法：




```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)
```

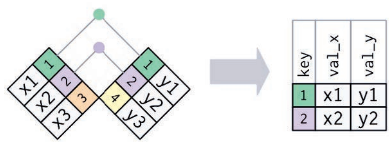
有颜色的列表示作为"键"的变量：它们用于在表间匹配行。灰色列表示"值"列，是与键对应的值。在以下的示例中，虽然键和值都是一个变量，但非常容易推广到多个键变量和多个值变量的情况。

连接是将 x 中每行连接到 y 中 0 行、一行或多行的一种方法。下图表示出了所有可能的匹配，匹配就是两行之间的交集。



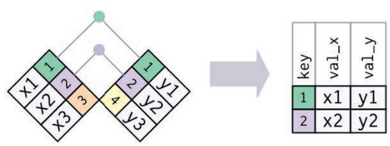
（如果观察得足够仔细，那么你就会发现我们交换了 x 中的键列和值列的顺序。这只是为了强调连接是按照键来进行匹配的。实际上键和值的对应关系没有改变。）

匹配在实际的连接操作中是用圆点表示的。圆点的数量 = 匹配的数量 = 结果中行的数量。



9.4.2 内连接

内连接是最简单的一种连接。只要两个观测的键是相等的，内连接就可以匹配它们。



（确切地说，这是一种内部**等值连接**，因为在匹配键时使用的是等值运算符。因为多数连接都是等值连接，所以我们通常省略这种说明。）

内连接的结果是一个新数据框，其中包含键、x 值和 y 值。我们使用 by 参数告诉 dplyr 哪个变量是键：

```
x %>%
  inner_join(y, by = "key")
#> # A tibble: 2 × 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1    x1    y1
#> 2     2    x2    y2
```

内连接最重要的性质是，没有匹配的行不会包含在结果中。这意味着内连接一般不适合在分析中使用，因为太容易丢失观测了。

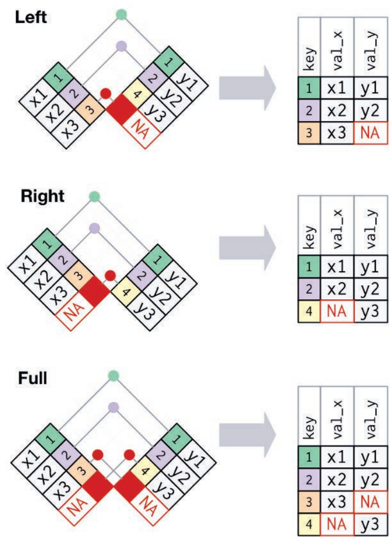
9.4.3 外连接

内连接保留同时存在于两个表中的观测，**外连接**则保留至少存在于一个表中的观测。外连接有 3 种类型。

- **左连接**：保留 x 中的所有观测。
- **右连接**：保留 y 中的所有观测
- **全连接**：保留 x 和 y 中的所有观测。

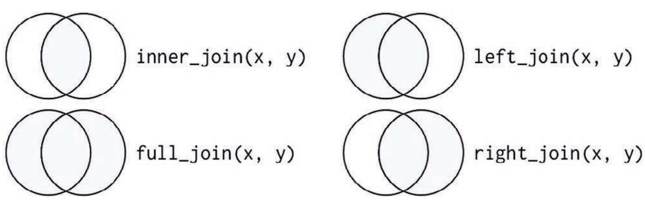
这些连接会向每个表中添加额外的“虚拟”观测，这个观测拥有总是匹配的键（如果没有其他键可匹配的话），其值则用 NA 来填充。

图形表示如下所示。



最常用的连接是左连接：只要想从另一张表中添加数据，就可以使用左连接，因为它会保留原表中的所有观测，即使它没有匹配。左连接应该是你的默认选择，除非有足够充分的理由选择其他的连接方式。

表示不同类型连接的另一种方式是使用维恩图。

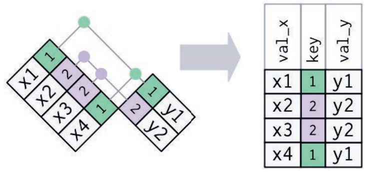


但是这并不是一种非常好的表示方式。虽然可以说明哪种连接会保留哪个表中的观测，但它具有非常明显的局限性：当键不能唯一标识观测时，维恩图无法表示这种情况。

9.4.4 重复键

至今为止，所有图都假设键具有唯一性。但情况并非总是如此。本节说明了当键不唯一时将会发生的两种情况。

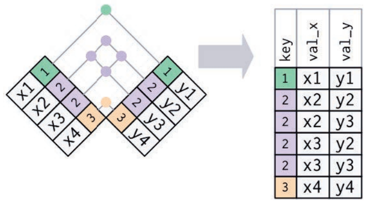
- 一张表中具有重复键。通常来说，当存在一对多关系时，如果你想要向表中添加额外信息，就会出现这种情况。



注意，我们稍稍调整了键列在结果中的位置，这样可以反映出这个键是 y 的主键、x 的外键：

```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     2, "x3",
     1, "x4"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2"
)
left_join(x, y, by = "key")
#> # A tibble: 4 × 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1    x1    y1
#> 2     2    x2    y2
#> 3     2    x3    y2
#> 4     1    x4    y1
```

- 两张表中都有重复键。这通常意味着出现了错误，因为键在任意一张表中都不能唯一标识观测。当连接这样的重复键时，你会得到所有可能的组合，即笛卡儿积：




```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     2, "x3",
     3, "x4"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     2, "y3",
```

```
    3, "y4"
)
left_join(x, y, by = "key")
#> # A tibble: 6 × 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1    x1    y1
#> 2     2    x2    y2
#> 3     2    x2    y3
#> 4     2    x3    y2
#> 5     2    x3    y3
#> 6     3    x4    y4
```

9.4.5 定义键列

迄今为止，两张表都是通过一个单变量来连接的，而且这个变量在两张表中具有同样的名称。这种限制条件是通过 `by = "key"` 来实现的。你还可以对 `by` 设置其他值，以另外的方式来连接表。

- 默认值 `by = NULL`。这会使用存在于两个表中的所有变量，这种方式称为**自然连接**。例如，匹配航班表和天气表时使用的就是其公共变量：`year`、`month`、`day`、`hour` 和 `origin`。

```
flights2 %>%
  left_join(weather)
#> Joining, by = c("year", "month", "day", "hour",
#>   "origin")
#> # A tibble: 336,776 × 18
#>    year month   day  hour origin  dest tailnum
#>   <dbl> <dbl> <int> <dbl>  <chr> <chr>   <chr>
#> 1  2013     1     1     5    EWR   IAH  N14228
#> 2  2013     1     1     5    LGA   IAH  N24211
#> 3  2013     1     1     5    JFK   MIA  N619AA
#> 4  2013     1     1     5    JFK   BQN  N804JB
#> 5  2013     1     1     6    LGA   ATL  N668DN
#> 6  2013     1     1     5    EWR   ORD  N39463
#> # ... with 3.368e+05 more rows, and 11 more variables:
#> #  carrier <chr>, temp <dbl>, dewp <dbl>,
#> #  humid <dbl>, wind_dir <dbl>, wind_speed <dbl>,
#> #  wind_gust <dbl>, precip <dbl>, pressure <dbl>,
#> #  visib <dbl>, time_hour <dttm>
```

- 字符向量 `by = "x"`。这种方式与自然连接很相似，但只使用某些公共变量。例如，`flights` 和 `planes` 表中都有 `year` 变量，但是它们的意义不同，因此我们只通过 `tailnum` 进行连接：

```
flights2 %>%
  left_join(planes, by = "tailnum")
#> # A tibble: 336,776 × 16
#>   year.x month   day  hour origin  dest tailnum
#>    <int> <int> <int> <dbl>  <chr> <chr>   <chr>
#> 1   2013     1     1     5    EWR   IAH  N14228
#> 2   2013     1     1     5    LGA   IAH  N24211
#> 3   2013     1     1     5    JFK   MIA  N619AA
```

```
#> 4   2013     1     1     5    JFK   BQN  N804JB
#> 5   2013     1     1     6    LGA   ATL  N668DN
#> 6   2013     1     1     5    EWR   ORD  N39463
#> # ... with 3.368e+05 more rows, and 9 more variables:
#> #   carrier <chr>, year.y <int>, type <chr>,
#> #   manufacturer <chr>, model <chr>, engines <int>,
#> #   seats <int>, speed <int>, engine <chr>
```

注意，结果中的 year 变量（同时存在于两个输入数据框中，但并不要求相等）添加了一个后缀，以消除歧义。

- 命名字符向量 by = c("a" = "b")。这种方式会匹配 x 表中的 a 变量和 y 表中的 b 变量。输出结果中使用的是 x 表中的变量。

  例如，如果想要画出一幅地图，那么我们就需要在航班数据中加入机场数据，后者包含了每个机场的位置（lat 和 lon）。因为每次航班都有起点机场和终点机场，所以需要指定使用哪个机场进行连接：

```
flights2 %>%
  left_join(airports, c("dest" = "faa"))
#> # A tibble: 336,776 × 14
#>    year month   day  hour origin  dest tailnum
#>   <int> <int> <int> <dbl>  <chr> <chr>   <chr>
#> 1  2013     1     1     5    EWR   IAH  N14228
#> 2  2013     1     1     5    LGA   IAH  N24211
#> 3  2013     1     1     5    JFK   MIA  N619AA
#> 4  2013     1     1     5    JFK   BQN  N804JB
#> 5  2013     1     1     6    LGA   ATL  N668DN
#> 6  2013     1     1     5    EWR   ORD  N39463
#> # ... with 3.368e+05 more rows, and 7 more variables:
#> #  carrier <chr>, name <chr>, lat <dbl>, lon <dbl>,
#> #  alt <int>, tz <dbl>, dst <chr>

flights2 %>%
  left_join(airports, c("origin" = "faa"))
#> # A tibble: 336,776 × 14
#>    year month   day  hour origin  dest tailnum
#>   <int> <int> <int> <dbl>  <chr> <chr>   <chr>
#> 1  2013     1     1     5    EWR   IAH  N14228
#> 2  2013     1     1     5    LGA   IAH  N24211
#> 3  2013     1     1     5    JFK   MIA  N619AA
#> 4  2013     1     1     5    JFK   BQN  N804JB
#> 5  2013     1     1     6    LGA   ATL  N668DN
#> 6  2013     1     1     5    EWR   ORD  N39463
#> # ... with 3.368e+05 more rows, and 7 more variables:
#> #   carrier <chr>, name <chr>, lat <dbl>, lon <dbl>,
#> #   alt <int>, tz <dbl>, dst <chr>
```

9.4.6 练习

(1) 计算出每个目的地的平均延误时间，然后与 airports 数据框连接，从而展示出延误的空间分布。以下是画出美国地图的一种简单方法。

```
airports %>%
  semi_join(flights, c("faa" = "dest")) %>%
  ggplot(aes(lon, lat)) +
    borders("state") +
    geom_point() +
    coord_quickmap()
```

（别担心不理解 semi_join() 的意义，下一节就会对其进行介绍。）

你可以使用数据点的 size 或 color 属性来表示每个机场的平均延误时间。

(2) 将起点机场和终点机场的位置信息（即 lat 和 lon）添加到 flights 中。

(3) 飞机的机龄和延误时间有关系吗？

(4) 什么样的天气状况更容易出现延误？

(5) 2013 年 6 月 13 日发生了什么情况？展示出这天延误时间的空间模式，并使用 Google 说明一下这天的天气状况。

9.4.7 其他实现方式

base::merge() 函数可以实现所有 4 种合并连接操作。

dplyr	merge
inner_join(x, y)	merge(x, y)
left_join(x, y)	merge(x, y, all.x = TRUE)
right_join(x, y)	merge(x, y, all.y = TRUE)
full_join(x, y)	merge(x, y, all.x = TRUE, all.y = TRUE)

dplyr 连接操作的优点是，可以更加清晰地表达出代码的意图：不同连接间的区别确实非常重要，但隐藏在 merge() 函数的参数中了。dplyr 连接操作的速度明显更快，而且不会弄乱行的顺序。

因为 SQL 是 dplyr 连接操作的灵感来源，所以二者之间的转换非常简单明了。

dplyr	SQL
inner_join(x, y, by = "z")	SELECT * FROM x INNER JOIN y USING (z)
left_join(x, y, by = "z")	SELECT * FROM x LEFT OUTER JOIN y USING (z)
right_join(x, y, by = "z")	SELECT * FROM x RIGHT OUTER JOIN y USING (z)
full_join(x, y, by = "z")	SELECT * FROM x FULL OUTER JOIN y USING (z)

注意，"INNER" 和 "OUTER" 是可选的，经常省略。

在表间连接不同变量（如 inner_join(x, y, by = c("a" = "b"))）时，SQL 的语法与以上有些区别：SELECT * FROM x INNER JOIN y ON x.a = y.b。从这种语法可以看出，与 dplyr 相比，SQL 支持的连接类型更广泛，因为 SQL 可以使用除相等关系外的其他逻辑关系来连接两个表（有时这称为非等值连接）。

9.5 筛选连接

筛选连接匹配观测的方式与合并连接相同，但前者影响的是观测，而不是变量。筛选连接有两种类型。

- `semi_join(x, y)`：**保留** x 表中与 y 表中的观测相匹配的所有观测。
- `anti_join(x, y)`：**丢弃** x 表中与 y 表中的观测相匹配的所有观测。

对数据表进行筛选或摘要统计后，如果想要使用表中原来的行来匹配筛选或摘要结果，那么半连接是非常有用的。例如，假设你已经找出了最受欢迎的前 10 个目的地：

```
top_dest <- flights %>%
  count(dest, sort = TRUE) %>%
  head(10)
top_dest
#> # A tibble: 10 × 2
#>    dest     n
#>   <chr> <int>
#> 1   ORD 17283
#> 2   ATL 17215
#> 3   LAX 16174
#> 4   BOS 15508
#> 5   MCO 14082
#> 6   CLT 14064
#> # ... with 4 more rows
```

现在想要找出飞往这些目的地的所有航班，你可以自己构造一个筛选器：

```
flights %>%
  filter(dest %in% top_dest$dest)
#> # A tibble: 141,145 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      542            540         2
#> 2  2013     1     1      554            600        -6
#> 3  2013     1     1      554            558        -4
#> 4  2013     1     1      555            600        -5
#> 5  2013     1     1      557            600        -3
#> 6  2013     1     1      558            600        -2
#> # ... with 1.411e+05 more rows, and 12 more variables:
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#> #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

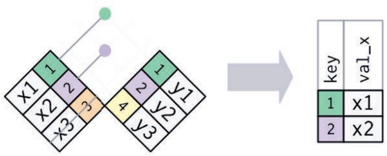
但这种方法很难扩展到多个变量。例如，假设已经找出了平均延误时间最长的 10 天，那么你应该如何使用 year、month 和 day 来构造筛选语句，才能在 `flights` 中找出这 10 天的观测？

此时你应该使用半连接，它可以像合并连接一样连接两个表，但不添加新列，而是保留 x 表中那些可以匹配 y 表的行：

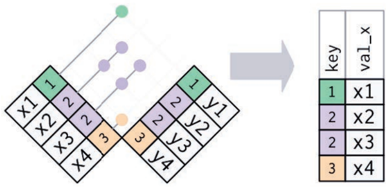
```
flights %>%
  semi_join(top_dest)
#> Joining, by = "dest"
#> # A tibble: 141,145 × 19
```

```
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      554            558        -4
#> 2  2013     1     1      558            600        -2
#> 3  2013     1     1      608            600         8
#> 4  2013     1     1      629            630        -1
#> 5  2013     1     1      656            700        -4
#> 6  2013     1     1      709            700         9
#> # ... with 1.411e+05 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#> #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

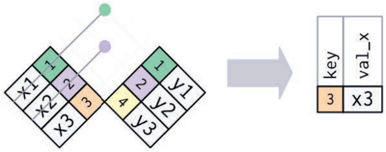
半连接的图形表示如下所示。



重要的是存在匹配，匹配了哪条观测则无关紧要。这说明筛选连接不会像合并连接那样造成重复的行。



半连接的逆操作是反连接。反连接保留 x 表中那些没有匹配 y 表的行。



反连接可以用于诊断连接中的不匹配。例如，在连接 `flights` 和 `planes` 时，你可能想知道 `flights` 中是否有很多行在 `planes` 中没有匹配记录：

```
flights %>%
  anti_join(planes, by = "tailnum") %>%
  count(tailnum, sort = TRUE)
#> # A tibble: 722 × 2
#>   tailnum     n
#>     <chr> <int>
#> 1    <NA>  2512
#> 2  N725MQ   575
#> 3  N722MQ   513
#> 4  N723MQ   507
#> 5  N713MQ   483
#> 6  N735MQ   396
#> # ... with 716 more rows
```

练习

(1) 如果一条航班信息的 `tailnum` 是缺失值，这说明什么？如果机尾编号在 `planes` 中没有匹配的记录，一般是什么情况？（提示：有一个变量可以解释约 90% 的这种情况。）

(2) 对航班信息进行筛选，只保留至少有 100 次飞行记录的飞机的航班信息。

(3) 使用 `fueleconomy::vehicles` 和 `fueleconomy::common` 找出那些用于最常用模型的记录。

(4) 找出这一整年中航班延误最严重的 48 小时。与 `weather` 数据互相参照，你能找出某种模式吗？

(5) 你能说出 `anti_join(flights, airports, by = c("dest" = "faa"))` 这条语句的意义吗？`anti_join(airports, flights, by = c("faa" = "dest"))` 这条语句的意义呢？

(6) 或许你认为飞机和航空公司之间存在着某种隐含关系，因为每架飞机都属于一个航空公司。使用你在前面章节中学到的工具来确认或否定这个假设。

9.6 连接中的问题

因为本章中所用的数据已经整理过了，所以使用时基本不会出现问题。在处理自己的数据时，感觉可不见得会有这么好。为了在使用自己的数据时可以顺畅地进行各种连接，你需要注意以下几点。

(1) 首先，需要找出每个表中可以作为主键的变量。一般应该基于对数据的理解来确定主键，而不是凭经验寻找能作为唯一标识符的变量组合。如果在确定主键时根本没有考虑过其意义，那么就可能步入歧途，虽然可以找出具有唯一性的变量组合，但它与数据间的关系却可能不是真实的。

例如，经度和纬度虽然能够唯一标识每个机场，但却不是良好的标识符！

```
airports %>% count(alt, lon) %>% filter(n > 1)
#> Source: local data frame [0 x 3]
#> Groups: alt [0]
#>
#> # ... with 3 variables: alt <int>, lon <dbl>, n <int>
```

(2) 确保主键中的每个变量都没有缺失值。如果有缺失值，那么这个变量就不能标识观测！

(3) 检查外键是否与另一张表的主键相匹配。最好的方法是使用 `anti_join()`，由于数据录入错误，外键和主键不匹配的情况很常见。解决这种问题通常需要大量工作。

如果键中确实有缺失值，那么你就要深思熟虑一下，是应该使用内连接还是外连接，此外，是否应该丢弃那些没有匹配记录的行。

注意，仅凭检查连接前后的行数是不足以确保连接能够顺畅运行的。如果进行了两张表都有重复键的内连接，那么就很可能不幸地遇到这种情况：被丢弃的行的数量正好等于重复行的数量！

9.7 集合操作

两表之间的最后一种操作就是集合操作。我们通常很少使用这种操作，但如果你想要将一个复杂的筛选操作分解为多个简单部分时，它们还是有些用处的。所有集合操作都是作用于整行的，比较的是每个变量的值。集合操作需要 x 和 y 具有相同的变量，并将观测按照集合来处理。

`intersect(x, y)`
: 返回既在 x 表，又在 y 表中的观测。

`union(x, y)`
: 返回 x 表或 y 表中的唯一观测。

`setdiff(x, y)`
: 返回在 x 表，但不在 y 表中的观测。

给定以下简单数据：

```
df1 <- tribble(
  ~x, ~y,
   1,  1,
   2,  1
)
df2 <- tribble(
  ~x, ~y,
   1,  1,
   1,  2
)
```

4 种可能的集合操作为：

```
intersect(df1, df2)
#> # A tibble: 1 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     1

# 注意，我们得到了3行，而不是4行
union(df1, df2)
#> # A tibble: 3 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     2
#> 2     2     1
#> 3     1     1

setdiff(df1, df2)
#> # A tibble: 1 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     2     1

setdiff(df2, df1)
#> # A tibble: 1 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     2
```

第 10 章
使用stringr处理字符串

10.1 简介

本章将介绍 R 中的字符串处理。你将学习字符串的基本工作原理，以及如何手工创建字符串，但本章的重点是**正则表达式**（regular expression，regexp）。正则表达式的用处非常大，字符串通常包含的是非结构化或半结构化数据，正则表达式可以用简练的语言来描述字符串中的模式。第一次见到正则表达式时，你可能会认为它是猫在键盘上踩出来的，但随着逐渐加深对它的理解后，你就能体会其中的深刻含义了。

准备工作

本章的重点是用于字符串处理的 stringr 包。因为不会一直处理文本数据，所以 stringr 不是 tidyverse 核心 R 包的一部分，我们需要使用命令来加载它。

```
library(tidyverse)
library(stringr)
```

10.2 字符串基础

可以使用单引号或双引号来创建字符串。与其他语言不同，单引号和双引号在 R 中没有区别。我们推荐使用 "，除非你想要创建包含多个 " 的一个字符串：

```
string1 <- "This is a string"
string2 <- 'To put a "quote" inside a string, use single quotes'
```

如果忘记了结尾的引号，你会看到一个 +，这是一个续行符：

```
> "This is a string without a closing quote
+
```

```
+
+ HELP I'M STUCK
```

如果遇到了这种情况，可以按 Esc 键，然后重新输入。

如果想要在字符串中包含一个单引号或双引号，可以使用 \ 对其进行“转义”：

```
double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'"
```

这意味着，如果想要在字符串中包含一个反斜杠，就需要使用两个反斜杠：\\。

注意，字符串的打印形式与其本身的内容不是相同的，因为打印形式中会显示出转义字符。如果想要查看字符串的初始内容，可以使用 `writelines()` 函数：

```
x <- c("\"", "\\")
x
#> [1] "\"" "\\"
writeLines(x)
#> "
#> \
```

还有其他几种特殊字符。最常用的是换行符 `\n` 和制表符 `\t`，你可以使用 `?'"'` 或 `?"'"` 调出帮助文件来查看完整的特殊字符列表。有时你还会看到 `"\u00b5"` 这样的字符串，这是一种在所有平台上都有效的非英文字符的写法：

```
x <- "\u00b5"
x
#> [1] "µ"
```

多个字符串通常保存在一个字符向量中，你可以使用 `c()` 函数来创建字符向量：

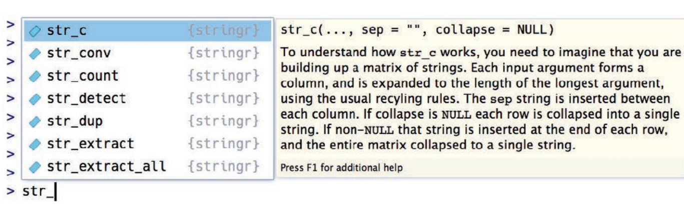
```
c("one", "two", "three")
#> [1] "one"   "two"   "three"
```

10.2.1 字符串长度

R 基础包中包含了很多字符串处理函数，但我们尽量不使用这些函数，因为它们的使用方法不一致，很难记忆。相反，我们将使用 stringr 中的函数，这些函数的名称更直观，并且都是以 `str_` 开头的。例如，`str_length()` 函数可以返回字符串中的字符数量：

```
str_length(c("a", "R for data science", NA))
#> [1]  1 18 NA
```

如果使用 RStudio，那么通用前缀 `str_` 会特别有用，因为输入 `str_` 后会触发自动完成功能，你可以看到所有的字符串函数：



10.2.2 字符串组合

要想组合两个或更多字符串，可以使用 str_c() 函数：

```
str_c("x", "y")
#> [1] "xy"
str_c("x", "y", "z")
#> [1] "xyz"
```

可以使用 sep 参数来控制字符串间的分隔方式：

```
str_c("x", "y", sep = ", ")
#> [1] "x, y"
```

和多数 R 函数一样，缺失值是可传染的。如果想要将它们输出为 "NA"，可以使用 str_replace_na()：

```
x <- c("abc", NA)
str_c("|-", x, "-|")
#> [1] "|-abc-|" NA
str_c("|-", str_replace_na(x), "-|")
#> [1] "|-abc-|" "|-NA-|"
```

如以上代码所示，str_c() 函数是向量化的，它可以自动循环短向量，使得其与最长的向量具有相同的长度：

```
str_c("prefix-", c("a", "b", "c"), "-suffix")
#> [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

长度为 0 的对象会被无声无息地丢弃。这与 if 结合起来特别有用：

```
name <- "Hadley"
time_of_day <- "morning"
birthday <- FALSE

str_c(
  "Good ", time_of_day, " ", name,
  if (birthday) " and HAPPY BIRTHDAY",
  "."
)
#> [1] "Good morning Hadley."
```

要想将字符向量合并为字符串，可以使用 collapse() 函数：

```
str_c(c("x", "y", "z"), collapse = ", ")
#> [1] "x, y, z"
```

10.2.3 字符串取子集

可以使用 str_sub() 函数来提取字符串的一部分。除了字符串参数外，str_sub() 函数中还有 start 和 end 参数，它们给出了子串的位置（包括 start 和 end 在内）：

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
```

```
#> [1] "App" "Ban" "Pea"

# 负数表示从后往前数
str_sub(x, -3, -1)
#> [1] "ple" "ana" "ear"
```

注意，即使字符串过短，`str_sub()` 函数也不会出错，它将返回尽可能多的字符：

```
str_sub("a", 1, 5)
#> [1] "a"
```

还可以使用 `str_sub()` 函数的赋值形式来修改字符串：

```
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
#> [1] "apple"  "banana" "pear"
```

10.2.4 区域设置

前面我们使用了 `str_to_lower()` 函数将文本转换为小写，你还可以使用 `str_to_upper()` 或 `str_to_title()` 函数。但是，大小写转换要比你想象的更复杂，因为不同的语言有不同的转换规则。你可以通过明确区域设置来选择使用哪种规则：

```
# 土耳其语中有带点和不带点的两个i，它们在转换为大写时是不同的：
str_to_upper(c("i", "ı"))
#> [1] "I" "I"
str_to_upper(c("i", "ı"), locale = "tr")
#> [1] "İ" "I"
```

区域设置可以参考 ISO 639 语言编码标准，语言编码是 2 或 3 个字母的缩写。如果不知道自己的语言编码，那么你可以查看维基百科上的 List of ISO 639-1 codes，其中有一个非常好的列表。如果没有进行区域设置，那么函数就会使用由操作系统提供的当前区域设置。

受区域设置影响的另一种重要操作是排序。R 基础包中的 `order()` 和 `sort()` 函数使用当前区域设置对字符串进行排序。如果需要更强大的、可以在不同计算机间使用的排序操作，那么可以使用 `str_sort()` 和 `str_order()` 函数，它们可以使用 `locale` 参数来进行区域设置：

```
x <- c("apple", "eggplant", "banana")

str_sort(x, locale = "en")  # 英语
#> [1] "apple"    "banana"   "eggplant"

str_sort(x, locale = "haw") # 夏威夷语
#> [1] "apple"    "eggplant" "banana"
```

10.2.5 练习

(1) 在没有使用 stringr 的那些代码中，你会经常看到 `paste()` 和 `paste0()` 函数，这两个函数的区别是什么？stringr 中的哪两个函数与它们是对应的？这些函数处理 `NA` 的方式有什么不同？

(2) 用自己的语言描述一下 str_c() 函数的 sep 和 collapse 参数有什么区别？

(3) 使用 str_length() 和 str_sub() 函数提取出一个字符串最中间的字符。如果字符串中的字符数是偶数，你应该怎么做？

(4) str_wrap() 函数的功能是什么？应该在何时使用这个函数？

(5) str_trim() 函数的功能是什么？其逆操作是哪个函数？

(6) 编写一个函数将字符向量转换为字符串，例如，将字符向量 c("a", "b", "c") 转换为字符串 a、b 和 c。仔细思考一下，如果给定一个长度为 0、1 或 2 的向量，那么这个函数应该怎么做？

10.3　使用正则表达式进行模式匹配

正则表达式是一门非常精练的语言，可以描述字符串中的模式。理解正则表达式需要花费一点精力，但是一旦理解了，你就会发现其功能如此强大。

我们通过 str_view() 和 str_view_all() 函数来学习正则表达式。这两个函数接受一个字符向量和一个正则表达式，并显示出它们是如何匹配的。我们先从非常简单的正则表达式开始，然后循序渐进地学习更加复杂的正则表达式。一旦掌握了模式匹配，你就知道如何将这种思想应用于不同的 stringr 函数了。

10.3.1　基础匹配

最简单的模式是精确匹配字符串[1]：

```
x <- c("apple", "banana", "pear")
str_view(x, "an")
```

```
apple
banana
pear
```

另一个更复杂一些的模式是使用 .，它可以匹配任意字符（除了换行符）：

```
str_view(x, ".a.")
```

```
apple
banana
pear
```

但是，如果 . 可以匹配任意字符，那么如何匹配字符 . 呢？你需要使用一个“转义”符号来告诉正则表达式实际上就是要匹配 . 这个字符，而不是使用 . 来匹配其他字符。和字符串一样，正则表达式也使用反斜杠来去除某些字符的特殊含义。因此，如果要匹配 .，那么你需要的正则表达式就是 \.。遗憾的是，这样做会带来一个问题。因为我们使用字符串来表示正则表达式，而且 \ 在字符串中也用作转义字符，所以正则表达式 \. 的字符串形式应是 \\.：

注 1：要想显示以下图片，还需要额外安装两个 R 包：htmltools 和 htmlwidgets。——译者注

```
# 要想建立正则表示式，我们需要使用\\
dot <- "\\."

# 实际上表达式本身只包含一个\:
writeLines(dot)
#> \.

# 这个表达式告诉R搜索一个.
str_view(c("abc", "a.c", "bef"), "a\\.c")
```

abc
a.c
bef

如果 \ 在正则表达式中用作转义字符，那么如何匹配 \ 这个字符呢？我们还是需要去除其特殊意义，建立形式为 \\ 的正则表达式。要想建立这样的正则表达式，我们需要使用一个字符串，其中还需要对 \ 进行转义。这意味着要想匹配字符 \，我们需要输入 "\\\\"——你需要 4 个反斜杠来匹配 1 个反斜杠！

```
x <- "a\\b"
writeLines(x)
#> a\b

str_view(x, "\\\\")
```

a\b

本书将正则表达式写作 \.，将表示正则表达式的字符串写作 "\\."。

10.3.2 练习

(1) 解释一下为什么这些字符串不能匹配一个反斜杠 \："\"、"\\"、"\\\"。

(2) 如何匹配字符序列 "'\ ？

(3) 正则表达式 \..\..\.. 会匹配哪种模式？如何用字符串来表示这个正则表达式？

10.3.3 锚点

默认情况下，正则表达式会匹配字符串的任意部分。有时我们需要在正则表达式中设置**锚点**，以便 R 从字符串的开头或末尾进行匹配。我们可以设置两种锚点。

- ^ 从字符串开头进行匹配。
- $ 从字符串末尾进行匹配。

```
x <- c("apple", "banana", "pear")
str_view(x, "^a")
```

apple
banana
pear

```
str_view(x, "a$")
```

```
apple
banana
pear
```

为了帮助你记住并区分这两种锚点的用法，我们介绍一种来自于 Evan Misshula 的记忆方法：始于权力（^），终于金钱（$）[2]。

如果想要强制正则表达式匹配一个完整字符串，那么可以同时设置 ^ 和 $ 这两个锚点：

```
x <- c("apple pie", "apple", "apple cake")
str_view(x, "apple")
```

```
apple pie
apple
apple cake
```

```
str_view(x, "^apple$")
```

```
apple pie
apple
apple cake
```

还可以使用 \b 来匹配单词间的边界。这种匹配在 R 中不是很常用，但当我们想在 RStudio 中搜索可以用于其他函数中的函数名称时，有时会使用这种方式。例如，为了避免匹配到 summarize、summary、rowsum 等，我们会使用 \bsum\b 进行搜索。

10.3.4 练习

(1) 如何匹配字符串 "$^$"？

(2) 给定 stringr::words 中的常用单词语料库，创建正则表达式以找出满足下列条件的所有单词。

a. 以 y 开头的单词。
b. 以 x 结尾的单词。
c. 长度正好为 3 个字符的单词。（不要使用 str_length() 函数，这是作弊！）
d. 具有 7 个或更多字符的单词。
因为这个列表非常长，所以你可以设置 str_view() 函数的 match 参数，只显示匹配的单词（match = TRUE）或未匹配的单词（match = FALSE）。

10.3.5 字符类与字符选项

很多特殊模式可以匹配多个字符。我们已经介绍过 .，它可以匹配除换行符外的任意字符。还有其他 4 种常用的字符类。

- \d 可以匹配任意数字。

注 2：^ 经常用于表示乘方运算，乘方和权力在英文中都是 power 这个单词。——译者注

- \s 可以匹配任意空白字符（如空格、制表符和换行符）。
- [abc] 可以匹配 a、b 或 c。
- [^abc] 可以匹配除 a、b、c 外的任意字符。

请牢记，要想创建包含 \d 或 \s 的正则表达式，你需要在字符串中对 \ 进行转义，因此需要输入 "\\d" 或 "\\s"。

你还可以使用**字符选项**创建多个可选的模式。例如，abc|d..f 可以匹配 abc 或 deaf。注意，因为 | 的优先级很低，所以 abc|xyz 匹配的是 abc 或 xyz，而不是 abcyz 或 abxyz。与数学表达式一样，如果优先级让人感到困惑，那么可以使用括号让其表达得更清晰一些：

```
str_view(c("grey", "gray"), "gr(e|a)y")
```

```
grey
gray
```

10.3.6 练习

(1) 创建正则表达式来找出符合以下条件的所有单词。

a. 以元音字母开头的单词。
b. 只包含辅音字母的单词（提示：考虑一下匹配“非”元音字母）。
c. 以 ed 结尾，但不以 eed 结尾的单词。
d. 以 ing 或 ize 结尾的单词。

(2) 实际验证一下规则：i 总是在 e 前面，除非 i 前面有 c。

(3) q 后面总是跟着一个 u 吗？

(4) 编写一个正则表达式来匹配英式英语单词，排除美式英语单词。

(5) 创建一个正则表达式来匹配你所在国家的电话号码。

10.3.7 重复

正则表达式的另一项强大功能是，其可以控制一个模式能够匹配多少次。

- ?：0 次或 1 次。
- +：1 次或多次。
- *：0 次或多次。

```
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, "CC?")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, "CC+")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, 'C[LX]+')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

注意，这些运算符的优先级非常高，因此使用 colou?r 既可以匹配 color，也可以匹配 colour。这意味着很多时候需要使用括号，比如 bana(na)+。

你还可以精确设置匹配的次数。

- {n}：匹配 *n* 次。
- {n,}：匹配 *n* 次或更多次。
- {,m}：最多匹配 *m* 次。
- {n, m}：匹配 *n* 到 *m* 次。

```
str_view(x, "C{2}")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, "C{2,}")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, "C{2,3}")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

默认的匹配方式是“贪婪的”：正则表达式会匹配尽量长的字符串。通过在正则表达式后面添加一个 ?，你可以将匹配方式更改为“懒惰的”，即匹配尽量短的字符串。虽然这是正则表达式的高级特性，但知道这一点是非常有用的。

```
str_view(x, 'C{2,3}?')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, 'C[LX]+?')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

10.3.8　练习

(1) 给出与 ?、+ 和 * 等价的 {m, n} 形式的正则表达式。

(2) 用语言描述以下正则表达式匹配的是何种模式（仔细阅读来确认我们使用的是正则表达式，还是定义正则表达式的字符串）？

a. ^.*$
b. "\\{.+\\}"
c. \d{4}-\d{2}-\d{2}
d. "\\\\{4}"

(3) 创建正则表达式来找出满足以下条件的所有单词。

a. 以 3 个辅音字母开头的单词。

b. 有连续 3 个或更多元音字母的单词。
c. 有连续 2 个或更多元音—辅音配对的单词。

(4) 解一下 https://regexcrossword.com/challenges/beginner 中的正则表达式入门级纵横字谜。

10.3.9 分组与回溯引用

你已经在前面学习了括号可以用于消除复杂表达式中的歧义。括号还可以定义“分组”，你可以通过**回溯引用**（如 \1、\2 等）来引用这些分组。例如，以下的正则表达式可以找出名称中有重复的一对字母的所有水果：

```
str_view(fruit, "(..)\\1", match = TRUE)
```

```
banana
coconut
cucumber
jujube
papaya
salal berry
```

（你很快就会看到这种匹配方式与 str_match() 函数结合起来使用是多么有效。）

10.3.10 练习

(1) 用语言描述以下正则表达式会匹配何种模式？

a. `(.)\1\1`
b. `"(.)(.)\\2\\1"`
c. `(..)\1`
d. `"(.).\\1.\\1"`
e. `"(.)(.)(.).*\\3\\2\\1"`

(2) 创建正则表达式来匹配出以下单词。

a. 开头字母和结尾字母相同的单词。
b. 包含一对重复字母的单词（例如，church 中包含了重复的 ch）。
c. 包含一个至少重复 3 次的字母的单词（例如，eleven 中的 e 重复了 3 次）。

10.4 工具

既然我们已经掌握了正则表达式的基础知识，现在是时候学习如何应用它们来解决实际问题了。我们将在本节中学习多种 stringr 函数，它们可以：

- 确定与某种模式相匹配的字符串；
- 找出匹配的位置；
- 提取出匹配的内容；
- 使用新值替换匹配内容；
- 基于匹配拆分字符串。

继续以下内容前，我们需要先提醒你一下：因为正则表达式太强大了，所以我们很容易认为所有问题都可以使用一个正则表达式来解决。正如 Jamie Zawinski 下面所说。

> 当遇到一个问题时，有些人会这样想：“我可以用正则表达式来搞定它。”于是，原来的一个问题就变成了两个问题。

作为一则警世恒言，我们看看以下检查电子邮件地址是否有效的这个正则表达式：

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\
[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\
\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:
[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\
[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:
\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\
["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\
\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\
["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\
\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:
(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\
[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:
(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?
[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:
[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:
(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\
[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:
(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:
\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?
[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\
\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?
[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\
\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:
(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\
[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".
\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?
[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\
["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)(?:,\s*(?:(?:
[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:
(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?
[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\
\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|
(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:
(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:
\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\
[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?
[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\
\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:
(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\
[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".
\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?
[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\
["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*))*)?;\s*)
```

这个示例是不是有些变态（当然电子邮件的地址确实特别复杂），但它确实来自于实际代码。可以查看 Stack Overflow 上的讨论（http://stackoverflow.com/a/201378）来获取更多信息。

别忘了你使用的是一门编程语言，还有很多其他工具随时待命。相对于创建一个复杂的正则表达式，更简单的方式是创建多个简单的正则表达式。如果很难使用一个正则表达式来解决问题，那么你就应该回过头仔细思考一下，是否可以将一个问题分解为多个小问题，然后一个个地依次解决。

10.4.1 匹配检测

要想确定一个字符向量能否匹配一种模式，可以使用 `str_detect()` 函数。它返回一个与输入向量具有同样长度的逻辑向量：

```
x <- c("apple", "banana", "pear")
str_detect(x, "e")
#> [1]  TRUE FALSE TRUE
```

记住，从数学意义上来说，逻辑向量中的 `FALSE` 为 0，`TRUE` 为 1。这使得在匹配特别大的向量时，`sum()` 和 `mean()` 函数能够发挥更大的作用：

```
# 有多少个以t开头的常用单词?
sum(str_detect(words, "^t"))
#> [1] 65
# 以元音字母结尾的常用单词的比例是多少?
mean(str_detect(words, "[aeiou]$"))
#> [1] 0.277
```

当逻辑条件非常复杂时（例如，匹配 a 或 b，但不匹配 c，除非 d 成立），一般来说，相对于创建单个正则表达式，使用逻辑运算符将多个 `str_detect()` 调用组合起来会更容易。例如，以下两种方法均可找出不包含元音字母的所有单词：

```
# 找出至少包含一个元音字母的所有单词，然后取反
no_vowels_1 <- !str_detect(words, "[aeiou]")
# 找出仅包含辅音字母（非元音字母）的所有单词
no_vowels_2 <- str_detect(words, "^[^aeiou]+$")
identical(no_vowels_1, no_vowels_2)
#> [1] TRUE
```

两种方法的结果是一样的，但我们认为第一种方法明显更容易理解。如果正则表达式过于复杂，则应该将其分解为几个更小的子表达式，将每个子表达式的匹配结果赋给一个变量，并使用逻辑运算组合起来。

`str_detect()` 函数的一种常见用法是选取出匹配某种模式的元素。你可以通过逻辑取子集方式来完成这种操作，也可以使用便捷的 `str_subset()` 包装器函数：

```
words[str_detect(words, "x$")]
#> [1] "box" "sex" "six" "tax"
str_subset(words, "x$")
#> [1] "box" "sex" "six" "tax"
```

然而，字符串通常会是数据框的一列，此时我们可以使用 `filter` 操作：

```
df <- tibble(
  word = words,
  i = seq_along(word)
)
df %>%
  filter(str_detect(words, "x$"))
#> # A tibble: 4 × 2
#>    word     i
#>   <chr> <int>
```

```
#> 1   box   108
#> 2   sex   747
#> 3   six   772
#> 4   tax   841
```

str_detect() 函数的一种变体是 str_count()，后者不是简单地返回是或否，而是返回字符串中匹配的数量：

```
x <- c("apple", "banana", "pear")
str_count(x, "a")
#> [1] 1 3 1

# 平均来看，每个单词中有多少个元音字母？
mean(str_count(words, "[aeiou]"))
#> [1] 1.99
```

str_count() 也完全可以同 mutate() 函数一同使用：

```
df %>%
  mutate(
    vowels = str_count(word, "[aeiou]"),
    consonants = str_count(word, "[^aeiou]")
  )
#> # A tibble: 980 × 4
#>       word     i vowels consonants
#>      <chr> <int>  <int>      <int>
#> 1        a     1      1          0
#> 2     able     2      2          2
#> 3    about     3      3          2
#> 4 absolute     4      4          4
#> 5   accept     5      2          4
#> 6  account     6      3          4
#> # ... with 974 more rows
```

abababa

注意，匹配从来不会重叠。例如，在 "abababa" 中，模式 "aba" 会匹配多少次？正则表达式会告诉你是 2 次，而不是 3 次：

```
str_count("abababa", "aba")
#> [1] 2
str_view_all("abababa", "aba")
```

abababa

注意 str_view_all() 函数的使用。你很快就会知道，很多 stringr 函数都是成对出现的：一个函数用于单个匹配，另一个函数用于全部匹配，后者会有后缀 _all。

10.4.2 练习

试着使用两种方法来解决以下每个问题，一种方法是使用单个正则表达式，另一种方法是使用多个 str_detect() 函数的组合。

a. 找出以 x 开头或结尾的所有单词。

b. 找出以元音字母开头并以辅音字母结尾的所有单词。
c. 是否存在包含所有元音字母的单词？
d. 哪个单词包含最多数量的元音字母？哪个单词包含最大比例的元音字母？（提示：分母应该是什么？）

10.4.3　提取匹配内容

要想提取匹配的实际文本，我们可以使用 `str_extract()` 函数。为了说明这个函数的用法，我们需要一个更加复杂的示例。我们将使用维基百科上的 Harvard sentences，这个数据集是用来测试 VOIP 系统的，但也可以用来练习正则表达式。这个数据集的全名是 `stringr::sentences`：

```
length(sentences)
#> [1] 720
head(sentences)
#> [1] "The birch canoe slid on the smooth planks."
#> [2] "Glue the sheet to the dark blue background."
#> [3] "It's easy to tell the depth of a well."
#> [4] "These days a chicken leg is a rare dish."
#> [5] "Rice is often served in round bowls."
#> [6] "The juice of lemons makes fine punch."
```

假设我们想要找出包含一种颜色的所有句子。首先，我们需要创建一个颜色名称向量，然后将其转换成一个正则表达式：

```
colors <- c(
  "red", "orange", "yellow", "green", "blue", "purple"
)
color_match <- str_c(colors, collapse = "|")
color_match
#> [1] "red|orange|yellow|green|blue|purple"
```

现在我们可以选取出包含一种颜色的句子，再从中提取出颜色，就可以知道有哪些颜色了：

```
has_color <- str_subset(sentences, color_match)
matches <- str_extract(has_color, color_match)
head(matches)
#> [1] "blue" "blue" "red"  "red"  "red"  "blue"
```

注意，`str_extract()` 只提取第一个匹配。我们可以先选取出具有多于一种匹配的所有句子，然后就可以很容易地看到更多匹配：

```
more <- sentences[str_count(sentences, color_match) > 1]
str_view_all(more, color_match)
```

```
It is hard to erase blue or red ink.
The green light in the brown box flickered.
The sky in the west is tinged with orange red.
```

```
str_extract(more, color_match)
#> [1] "blue"   "green"  "orange"
```

这是 stringr 函数的一种通用模式，因为单个匹配可以使用更简单的数据结构。要想得到所有匹配，可以使用 `str_extract_all()` 函数，它会返回一个列表：

```
str_extract_all(more, color_match)
#> [[1]]
#> [1] "blue" "red"
#>
#> [[2]]
#> [1] "green" "red"
#>
#> [[3]]
#> [1] "orange" "red"
```

你将在 15.5 节和第 16 章中学到更多关于列表的知识。

如果设置了 `simplify = TRUE`，那么 `str_extract_all()` 会返回一个矩阵，其中较短的匹配会扩展到与最长的匹配具有同样的长度：

```
str_extract_all(more, color_match, simplify = TRUE)
#>      [,1]     [,2]
#> [1,] "blue"   "red"
#> [2,] "green"  "red"
#> [3,] "orange" "red"

x <- c("a", "a b", "a b c")
str_extract_all(x, "[a-z]", simplify = TRUE)
#>      [,1] [,2] [,3]
#> [1,] "a"  ""   ""
#> [2,] "a"  "b"  ""
#> [3,] "a"  "b"  "c"
```

10.4.4　练习

(1) 在前面的示例中，你或许已经发现正则表达式匹配了 flickered，这并不是一种颜色。修改正则表达式来解决这个问题。

(2) 从 Harvard sentences 数据集中提取以下内容。

a. 每个句子的第一个单词。
b. 以 `ing` 结尾的所有单词。
c. 所有复数形式的单词。

10.4.5　分组匹配

我们在本章前面讨论了括号在正则表达式中的用法，它可以阐明优先级，还能对正则表达式进行分组，分组可以在匹配时回溯引用。你还可以使用括号来提取一个复杂匹配的各个部分。举例来说，假设我们想从句子中提取出名词。我们先进行一种启发式实验，找出跟在 a 或 the 后面的所有单词。因为使用正则表达式定义“单词”有一点难度，所以我们使用一种简单的近似定义——至少有 1 个非空格字符的字符序列：

```
noun <- "(a|the) ([^ ]+)"

has_noun <- sentences %>%
  str_subset(noun) %>%
  head(10)
has_noun %>%
  str_extract(noun)
#>  [1] "the smooth" "the sheet"  "the depth"  "a chicken"
#>  [5] "the parked" "the sun"    "the huge"   "the ball"
#>  [9] "the woman"  "a helps"
```

str_extract() 函数可以给出完整匹配；str_match() 函数则可以给出每个独立分组。str_match() 返回的不是字符向量，而是一个矩阵，其中一列是完整匹配，后面的列是每个分组的匹配：

```
has_noun %>%
  str_match(noun)
#>       [,1]         [,2]  [,3]
#>  [1,] "the smooth" "the" "smooth"
#>  [2,] "the sheet"  "the" "sheet"
#>  [3,] "the depth"  "the" "depth"
#>  [4,] "a chicken"  "a"   "chicken"
#>  [5,] "the parked" "the" "parked"
#>  [6,] "the sun"    "the" "sun"
#>  [7,] "the huge"   "the" "huge"
#>  [8,] "the ball"   "the" "ball"
#>  [9,] "the woman"  "the" "woman"
#> [10,] "a helps"    "a"   "helps"
```

（不出所料，这种启发式名词检测的效果并不好，它还找出了一些形容词，比如 smooth 和 parked。）

如果数据是保存在 tibble 中的，那么使用 tidyr::extract() 会更容易。这个函数的工作方式与 str_match() 函数类似，只是要求为每个分组提供一个名称，以作为新列放在 tibble 中：

```
tibble(sentence = sentences) %>%
  tidyr::extract(
    sentence, c("article", "noun"), "(a|the) ([^ ]+)",
    remove = FALSE
  )
#> # A tibble: 720 × 3
#>                                     sentence article    noun
#> *                                      <chr>   <chr>   <chr>
#> 1  The birch canoe slid on the smooth planks.     the  smooth
#> 2 Glue the sheet to the dark blue background.     the   sheet
#> 3      It's easy to tell the depth of a well.     the   depth
#> 4    These days a chicken leg is a rare dish.       a chicken
#> 5        Rice is often served in round bowls.    <NA>    <NA>
#> 6       The juice of lemons makes fine punch.    <NA>    <NA>
#> # ... with 714 more rows
```

与 str_extract() 函数一样，如果想要找出每个字符串的所有匹配，你需要使用 str_match_all() 函数。

10.4.6 练习

(1) 找出跟在一个数词（one、two、three 等）后面的所有单词，提取出数词与后面的单词。

(2) 找出所有缩略形式，分别列出撇号前面和后面的部分。

10.4.7 替换匹配内容

`str_replace()` 和 `str_replace_all()` 函数可以使用新字符串替换匹配内容。最简单的应用是使用固定字符串替换匹配内容：

```
x <- c("apple", "pear", "banana")
str_replace(x, "[aeiou]", "-")
#> [1] "-pple"  "p-ar"   "b-nana"
str_replace_all(x, "[aeiou]", "-")
#> [1] "-ppl-"  "p--r"   "b-n-n-"
```

通过提供一个命名向量，使用 `str_replace_all()` 函数可以同时执行多个替换：

```
x <- c("1 house", "2 cars", "3 people")
str_replace_all(x, c("1" = "one", "2" = "two", "3" = "three"))
#> [1] "one house"    "two cars"     "three people"
```

除了使用固定字符串替换匹配内容，你还可以使用回溯引用来插入匹配中的分组。在下面的代码中，我们交换了第二个单词和第三个单词的顺序：

```
sentences %>%
  str_replace("([^ ]+) ([^ ]+) ([^ ]+)", "\\1 \\3 \\2") %>%
  head(5)
#> [1] "The canoe birch slid on the smooth planks."
#> [2] "Glue sheet the to the dark blue background."
#> [3] "It's to easy tell the depth of a well."
#> [4] "These a days chicken leg is a rare dish."
#> [5] "Rice often is served in round bowls."
```

10.4.8 练习

(1) 使用反斜杠替换字符串中的所有斜杠。

(2) 使用 `replace_all()` 函数实现 `str_to_lower()` 函数的一个简单版。

(3) 交换 `words` 中单词的首字母和末尾字母，其中哪些字符串仍然是个单词？

10.4.9 拆分

`str_split()` 函数可以将字符串拆分为多个片段。例如，我们可以将句子拆分成单词：

```
sentences %>%
  head(5) %>%
  str_split(" ")
#> [[1]]
#> [1] "The"     "birch"   "canoe"   "slid"    "on"      "the"
```

```
#> [7] "smooth"  "planks."
#>
#> [[2]]
#> [1] "Glue"        "the"          "sheet"       "to"
#> [5] "the"  "dark"        "blue"         "background."
#>
#> [[3]]
#> [1] "It's"  "easy"  "to"    "tell"  "the"   "depth" "of"
#> [8] "a"   "well."
#>
#> [[4]]
#> [1] "These"  "days"    "a"        "chicken" "leg"     "is"
#> [7] "a" "rare"    "dish."
#>
#> [[5]]
#> [1] "Rice"   "is"      "often"  "served" "in"      "round"
#> [7] "bowls."
```

因为字符向量的每个分量会包含不同数量的片段，所以 str_split() 会返回一个列表。如果你拆分的是长度为 1 的向量，那么只要简单地提取列表的第一个元素即可：

```
"a|b|c|d" %>%
  str_split("\\|") %>%
  .[[1]]
#> [1] "a" "b" "c" "d"
```

否则，和返回列表的其他 stringr 函数一样，你可以通过设置 simplify = TRUE 返回一个矩阵：

```
sentences %>%
  head(5) %>%
  str_split(" ", simplify = TRUE)
#>      [,1]    [,2]    [,3]    [,4]      [,5]  [,6]    [,7]
#> [1,] "The"   "birch" "canoe" "slid"    "on"  "the"   "smooth"
#> [2,] "Glue"  "the"   "sheet" "to"      "the" "dark"  "blue"
#> [3,] "It's"  "easy"  "to"    "tell"    "the" "depth" "of"
#> [4,] "These" "days"  "a"     "chicken" "leg" "is"    "a"
#> [5,] "Rice"  "is"    "often" "served"  "in"  "round" "bowls."
#>      [,8]          [,9]
#> [1,] "planks."     ""
#> [2,] "background." ""
#> [3,] "a"           "well."
#> [4,] "rare"        "dish."
#> [5,] ""            ""
```

你还可以设定拆分片段的最大数量：

```
fields <- c("Name: Hadley", "Country: NZ", "Age: 35")
fields %>% str_split(": ", n = 2, simplify = TRUE)
#>      [,1]      [,2]
#> [1,] "Name"    "Hadley"
#> [2,] "Country" "NZ"
#> [3,] "Age"     "35"
```

除了模式，你还可以通过字母、行、句子和单词边界（boundary() 函数）来拆分字符串：

```
x <- "This is a sentence. This is another sentence."
str_view_all(x, boundary("word"))
```

This is a sentence. This is another sentence.

```
str_split(x, " ")[[1]]
#> [1] "This"      "is"        "a"         "sentence." ""
#> [6] "This"
#> [7] "is"        "another"   "sentence."
str_split(x, boundary("word"))[[1]]
#> [1] "This"     "is"       "a"        "sentence" "This"
#> [6] "is"
#> [7] "another"  "sentence"
```

10.4.10 练习

(1) 拆分字符串 "apples, pears, and bananas"。

(2) 为什么使用 boundary("word") 的拆分效果要比 "" 好？

(3) 使用空字符串（""）进行拆分会得到什么结果？尝试一下，然后阅读文档。

10.4.11 定位匹配内容

str_locate() 和 str_locate_all() 函数可以给出每个匹配的开始位置和结束位置。当没有其他函数能够精确地满足需求时，这两个函数特别有用。你可以使用 str_locate() 函数找出匹配的模式，然后使用 str_sub() 函数来提取或修改匹配的内容。

10.5 其他类型的模式

当使用一个字符串作为模式时，R 会自动调用 regex() 函数对其进行包装：

```
# 正常调用：
str_view(fruit, "nana")
# 上面形式是以下形式的简写
str_view(fruit, regex("nana"))
```

你可以使用 regex() 函数的其他参数来控制具体的匹配方式。

- ignore_case = TRUE 既可以匹配大写字母，也可以匹配小写字母，它总是使用当前的区域设置：

```
bananas <- c("banana", "Banana", "BANANA")
str_view(bananas, "banana")
```

banana
Banana
BANANA

```
str_view(bananas, regex("banana", ignore_case = TRUE))
```

```
banana
Banana
BANANA
```

- multiline = TRUE 可以使得 ^ 和 $ 从每行的开头和末尾开始匹配，而不是从完整字符串的开头和末尾开始匹配：

```
x <- "Line 1\nLine 2\nLine 3"
str_extract_all(x, "^Line")[[1]]
#> [1] "Line"
str_extract_all(x, regex("^Line", multiline = TRUE))[[1]]
#> [1] "Line" "Line" "Line"
```

- comments = TRUE 可以让你在复杂的正则表达式中加入注释和空白字符，以便更易理解。匹配时会忽略空格和 # 后面的内容。如果想要匹配一个空格，你需要对其进行转义："\\ "：

```
phone <- regex("
  \\(?     # 可选的开括号
  (\\d{3}) # 地区编码
  [)- ]?   # 可选的闭括号、短划线或空格
  (\\d{3}) # 另外3个数字
  [ -]?    # 可选的空格或短划线
  (\\d{3}) # 另外3个数字
  ", comments = TRUE)

str_match("514-791-8141", phone)
#>      [,1]          [,2]  [,3]  [,4]
#> [1,] "514-791-814" "514" "791" "814"
```

- dotall = TRUE 可以使得 . 匹配包括 \n 在内的所有字符。

除了 regex()，你还可以使用其他 3 种函数。

- fixed() 函数可以按照字符串的字节形式进行精确匹配，它会忽略正则表达式中的所有特殊字符，并在非常低的层次上进行操作。这样可以让你不用进行那些复杂的转义操作，而且速度比普通正则表达式要快很多。从以下的微基准测试可以看出，在这个简单的示例中，它的速度差不多是普通正则表达式的 3 倍：

```
microbenchmark::microbenchmark(
  fixed = str_detect(sentences, fixed("the")),
  regex = str_detect(sentences, "the"),
  times = 20
)
#> Unit: microseconds
#>   expr min  lq mean median  uq max neval cld
#>  fixed 116 117  136    120 125 389    20   a
#>  regex 333 337  346    338 342 467    20    b
```

  在匹配非英语数据时，要慎用 fixed() 函数。它可能会出现问题，因为此时同一个字符经常有多种表达方式。例如，定义 á 的方式有两种：一种是单个字母 a，另一种是 a 加上重音符号：

```
a1 <- "\u00e1"
a2 <- "a\u0301"
c(a1, a2)
#> [1] "á" "á"
a1 == a2
#> [1] FALSE
```

这两个字母的意义相同，但因为定义方式不同，所以 fixed() 函数找不到匹配。然而，你可以使用接下来将要介绍的 coll() 函数，按照我们使用的字符比较规则来进行匹配：

```
str_detect(a1, fixed(a2))
#> [1] FALSE
str_detect(a1, coll(a2))
#> [1] TRUE
```

- coll() 函数使用**标准排序规则**来比较字符串，这在进行不区分大小写的匹配时是非常有效的。注意，可以在 coll() 函数中设置 locale 参数，以确定使用哪种规则来比较字符。遗憾的是，世界各地所使用的规则是不同的！

```
# 这意味着在进行不区分大小写的匹配时，还是需要知道不同规则之间的区别：
i <- c("I", "İ", "i", "ı")
i
#> [1] "I" "İ" "i" "ı"

str_subset(i, coll("i", ignore_case = TRUE))
#> [1] "I" "i"
str_subset(
  i,
  coll("i", ignore_case = TRUE, locale = "tr")
)
#> [1] "İ" "i"
```

fixed() 和 regex() 函数中都有 ignore_case 参数，但都无法选择区域设置，它们总是使用默认的区域设置。你可以使用以下代码查看默认区域设置（我们稍后会对 stringi 包进行更多介绍）：

```
stringi::stri_locale_info()
#> $Language
#> [1] "en"
#>
#> $Country
#> [1] "US"
#>
#> $Variant
#> [1] ""
#>
#> $Name
#> [1] "en_US"
```

coll() 函数的弱点是速度，因为确定哪些是相同字符的规则比较复杂，与 regex() 和 fixed() 函数相比，coll() 确实比较慢。

- 在介绍 str_split() 函数时，你已经知道可以使用 boundary() 函数来匹配边界。你还可以在其他函数中使用这个函数：

```
x <- "This is a sentence."
str_view_all(x, boundary("word"))
```

```
This is a sentence.
```

```
str_extract_all(x, boundary("word"))
#> [[1]]
#> [1] "This"     "is"       "a"        "sentence"
```

练习

(1) 如何找出包含 \ 的所有字符串？分别使用 regex() 和 fixed() 函数来完成这个任务。

(2) sentences 数据集中最常见的 5 个单词是什么？

10.6 正则表达式的其他应用

R 基础包中有两个常用函数，它们也可以使用正则表达式。

- apropos() 函数可以在全局环境空间中搜索所有可用对象。当不能确切想起函数名称时，这个函数特别有用：

```
apropos("replace")
#> [1] "%+replace%"  "replace"         "replace_na"
#> [4] "str_replace" "str_replace_all" "str_replace_na"
#> [7] "theme_replace"
```

- dir() 函数可以列出一个目录下的所有文件。dir() 函数的 patten 参数可以是一个正则表达式，此时它只返回与这个模式相匹配的文件名。例如，你可以使用以下代码返回当前目录中的所有 R Markdown 文件：

```
head(dir(pattern = "\\.Rmd$"))
#> [1] "communicate-plots.Rmd" "communicate.Rmd"
#> [3] "datetimes.Rmd" "EDA.Rmd"
#> [5] "explore.Rmd" "factors.Rmd"
```

（如果更喜欢使用 *.Rmd 这样的“通配符”，你可以通过 glob2rx() 函数将其转换为正则表达式。）

10.7 stringi

stringr 建立于 stringi 的基础之上。stringr 非常容易学习，因为它只提供了非常少的函数，这些函数是精挑细选的，可以完成大部分常用字符串操作功能。与 stringr 不同，stringi 的设计思想是尽量全面，几乎包含了我们可以用到的所有函数：stringi 中有 234 个函数，而 stringr 中只有 42 个。

如果你发现某些工作很难使用 stringr 来完成，那么可以考虑使用 stringi。因为这两个包中的函数的工作方式非常相似，所以你可以很自然地从 stringr 过渡到 stringi。主要区别是前缀：`str_` 与 `stri_`。

练习

(1) 找出可以完成以下操作的 stringi 函数。

a. 计算单词的数量。
b. 找出重复字符串。
c. 生成随机文本。

(2) 如何控制 `stri_sort()` 函数用来排序的语言设置？

第 11 章

使用forcats处理因子

11.1 简介

因子在 R 中用于处理分类变量。分类变量是在固定的已知集合中取值的变量。当想要以非字母表顺序显示字符向量时，也可以使用分类变量。

从历史上看，因子远比字符串更容易处理。因此，R 基础包中的很多函数都自动将字符串转换为因子。这意味着因子经常出现在并不真正适合它们的地方。好在你不用担心 tidyverse 中会出现这种问题，可以将注意力集中于因子能够真正发挥作用的问题。

如果想要了解更多有关因子的背景知识，我们推荐你阅读一下 Roger Peng 的文章“An unauthorized biography”，以及 Thomas Lumley 的文章“stringsAsFactors = <sigh>”。

准备工作

我们将使用 forcats 包来处理因子，这个包提供了能够处理**分类变量**（其实就是因子的另一种说法）的工具，其中还包括了处理因子的大量辅助函数。因为 forcats 不是 tidyverse 的核心 R 包，所以需要手动加载。

```
library(tidyverse)
library(forcats)
```

11.2 创建因子

假设我们想要创建一个记录月份的变量：

```
x1 <- c("Dec", "Apr", "Jan", "Mar")
```

使用字符串来记录月份有两个问题。

(1) 月份只有 12 个取值，如果输入错误，那么代码不会有任何反应。

```
x2 <- c("Dec", "Apr", "Jam", "Mar")
```

(2) 其对月份的排序没有意义。

```
sort(x1)
#> [1] "Apr" "Dec" "Jan" "Mar"
```

你可以通过使用因子来解决以上两个问题。要想创建一个因子，必须先创建有效**水平**的一个列表：

```
month_levels <- c(
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
)
```

现在你可以创建因子了：

```
y1 <- factor(x1, levels = month_levels)
y1
#> [1] Dec Apr Jan Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
sort(y1)
#> [1] Jan Mar Apr Dec
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

不在有效水平集合内的所有值都会自动转换为 `NA`：

```
y2 <- factor(x2, levels = month_levels)
y2
#> [1] Dec  Apr  <NA> Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

如果想要显示错误信息，那么你可以使用 `readr::parse_factor()` 函数：

```
y2 <- parse_factor(x2, levels = month_levels)
#> Warning: 1 parsing failure.
#> row col           expected actual
#>   3  -- value in level set    Jam
```

如果省略了定义水平的这个步骤，那么会将按字母顺序排序的数据作为水平：

```
factor(x1)
#> [1] Dec Apr Jan Mar
#> Levels: Apr Dec Jan Mar
```

有时你会想让因子的顺序与初始数据的顺序保持一致。在创建因子时，将水平设置为 `unique(x)`，或者在创建因子后再对其使用 `fct_inorder()` 函数，就可以达到这个目的：

```
f1 <- factor(x1, levels = unique(x1))
f1
#> [1] Dec Apr Jan Mar
#> Levels: Dec Apr Jan Mar

f2 <- x1 %>% factor() %>% fct_inorder()
```

```
f2
#> [1] Dec Apr Jan Mar
#> Levels: Dec Apr Jan Mar
```

如果想要直接访问因子的有效水平集合，那么可以使用 `levels()` 函数：

```
levels(f2)
#> [1] "Dec" "Apr" "Jan" "Mar"
```

11.3 综合社会调查

本章后面的内容将重点讨论 `forcats::gss_cat` 数据集，该数据集是综合社会调查数据的一份抽样，综合社会调查是美国芝加哥大学的独立研究组织 NORC 进行的一项长期美国社会调查。这项调查包括几千个问题，我们挑选了一些变量放在 `gss_cat` 数据集中，它们可以说明处理因子时经常遇到的一些问题：

```
gss_cat
#> # A tibble: 21,483 × 9
#>    year       marital   age   race        rincome
#>   <int>        <fctr> <int> <fctr>         <fctr>
#> 1  2000 Never married    26  White  $8000 to 9999
#> 2  2000      Divorced    48  White  $8000 to 9999
#> 3  2000       Widowed    67  White Not applicable
#> 4  2000 Never married    39  White Not applicable
#> 5  2000      Divorced    25  White Not applicable
#> 6  2000       Married    25  White $20000 - 24999
#> # ... with 2.148e+04 more rows, and 4 more variables:
```

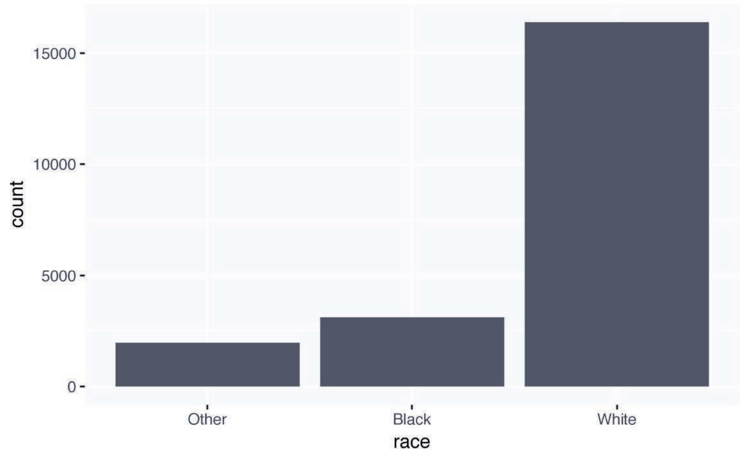
（记住，因为这个数据集是由一个 R 包提供的，所以你可以使用 `?gss_cat` 获取关于变量的更多信息。）

当因子保存在 tibble 中时，其水平不是很容易看到的。查看因子水平的一种方法是使用 `count()` 函数：

```
gss_cat %>%
  count(race)
#> # A tibble: 3 × 2
#>     race     n
#>   <fctr> <int>
#> 1  Other  1959
#> 2  Black  3129
#> 3  White 16395
```

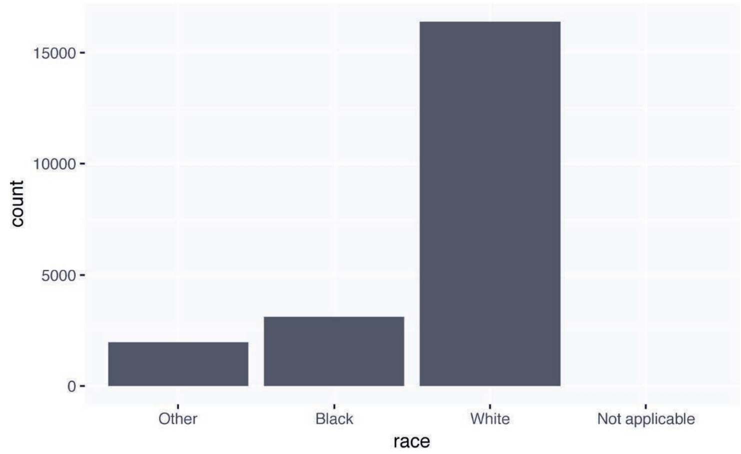
或者使用条形图：

```
ggplot(gss_cat, aes(race)) +
  geom_bar()
```



默认情况下，ggplot2 会丢弃没有任何数据的那些水平，你可以使用以下代码来强制显示这些水平：

```
ggplot(gss_cat, aes(race)) +
  geom_bar() +
  scale_x_discrete(drop = FALSE)
```



这些水平表示的也是有效值，只是没有出现在这个数据集中。遗憾的是，dplyr 中还没有 drop 这个选项，但很快就会有了。

在使用因子时，最常用的两种操作是修改水平的顺序和水平的值。下一节将详细介绍水平值的修改。

11.4　修改因子水平

比修改因子水平顺序更强大的操作是修改水平的值。修改水平不仅可以使得图形标签更美观清晰，以满足出版发行的要求，还可以将水平汇集成更高层次的显示。修改水平最常

用、最强大的工具是 fct_recode() 函数，它可以对每个水平进行修改或重新编码。例如，我们看一下 gss_cat$partyid：

```
gss_cat %>% count(partyid)
#> # A tibble: 10 × 2
#>                partyid     n
#>                 <fctr> <int>
#> 1           No answer   154
#> 2          Don't know     1
#> 3         Other party   393
#> 4  Strong republican  2314
#> 5 Not str republican  3032
#> 6        Ind,near rep  1791
#> # ... with 4 more rows
```

对水平的描述太过简单，而且不一致。我们将其修改为较为详细的排比结构：

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat"
  )) %>%
  count(partyid)
#> # A tibble: 10 × 2
#>                   partyid     n
#>                    <fctr> <int>
#> 1              No answer   154
#> 2             Don't know     1
#> 3            Other party   393
#> 4     Republican, strong  2314
#> 5       Republican, weak  3032
#> 6 Independent, near rep  1791
#> # ... with 4 more rows
```

fct_recode() 会让没有明确提及的水平保持原样，如果不小心修改了一个不存在的水平，那么它也会给出警告。

你可以将多个原水平赋给同一个新水平，这样就可以合并原来的分类：

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat",
    "Other"                 = "No answer",
    "Other"                 = "Don't know",
    "Other"                 = "Other party"
  )) %>%
```

```
  count(partyid)
#> # A tibble: 8 × 2
#>                partyid     n
#>                 <fctr> <int>
#> 1                Other   548
#> 2     Republican, strong  2314
#> 3       Republican, weak  3032
#> 4 Independent, near rep  1791
#> 5            Independent  4119
#> 6 Independent, near dem  2499
#> # ... with 2 more rows
```

使用这种操作时一定要小心：如果合并了原本不同的分类，那么就会产生误导性的结果。

如果想要合并多个水平，那么可以使用 `fct_recode()` 函数的变体 `fct_collapse()` 函数。对于每个新水平，你都可以提供一个包含原水平的向量：

```
gss_cat %>%
  mutate(partyid = fct_collapse(partyid,
    other = c("No answer", "Don't know", "Other party"),
    rep = c("Strong republican", "Not str republican"),
    ind = c("Ind,near rep", "Independent", "Ind,near dem"),
    dem = c("Not str democrat", "Strong democrat")
  )) %>%
  count(partyid)
#> # A tibble: 4 × 2
#>   partyid     n
#>    <fctr> <int>
#> 1   other   548
#> 2     rep  5346
#> 3     ind  8409
#> 4     dem  7180
```

练习

美国民主党、共和党和中间派的人数比例是如何随时间而变化的？

第 12 章

使用lubridate处理日期和时间

12.1 简介

本章将介绍如何在 R 中处理日期和时间。乍看起来，日期和时间非常容易。日常生活的每时每刻都在使用它们，而且它们似乎也不会引起很多混淆。但是，随着对日期和时间了解得越来越多，我们就会越来越发现其复杂之处。我们先热热身，思考以下 3 个似乎非常简单的问题。

- 每一年都是 365 天吗？
- 每一天都是 24 小时吗？
- 每分钟都是 60 秒吗？

你肯定知道不是每一年都是 365 天，但你知道确定某年是否为闰年的完整规则（包含 3 个部分）吗？你应该知道世界上很多地区使用夏时制，因此有些天是 23 个小时，有些天则是 25 个小时。但你可能根本不知道，有些分钟是 61 秒，因为地球自转正在逐渐变慢，所以我们有时候要增加 1 个闰秒。

日期和时间非常复杂，因为它们要兼顾两种物理现象（地球的自转以及围绕太阳的公转）和一系列地理政治现象（包括月份、时区和夏时制）。虽然本章不会介绍关于日期和时间的所有细节，但提供了处理日期和时间的实用技能的坚实基础，以帮助你解决常见的有关日期和时间的分析问题。

准备工作

本章主要讨论 lubridate 包，它可以使得 R 对日期和时间的处理更加容易。lubridate 不是 tidyverse 的核心 R 包，因为只有在处理日期和时间时才需要它。我们还需要将 nycflights13 包作为练习数据。

```
library(tidyverse)

library(lubridate)
library(nycflights13)
```

12.2 创建日期或时间

表示日期或时间的数据有 3 种类型。

- **日期**：在 tibble 中显示为 `<date>`。
- **时间**：一天中的某个时刻，在 tibble 中显示为 `<time>`。
- **日期时间**：可以唯一标识某个时刻（通常精确到秒）的日期加时间，在 tibble 中显示为 `<dttm>`。而这种类型在 R 语言的其他地方称为 POSIXct，但我们认为这个名称不是非常合适。

在本章中，我们将重点讨论日期和日期时间型数据，因为 R 中没有保存时间的原生类。如果需要处理时间数据，那么你可以使用 hms 包。

只要能够满足需要，你就应该使用最简单的数据类型。这意味着只要能够使用日期型数据，那么就不应该使用日期时间型数据。日期时间型数据要复杂得多，因为它需要处理时区，我们会在本章末尾继续讨论这个问题。

要想得到当前日期或当前日期时间，你可以使用 `today()` 或 `now()` 函数：

```
today()
#> [1] "2016-10-10"
now()
#> [1] "2016-10-10 15:19:39 PDT"
```

除此之外，以下 3 种方法也可以创建日期或时间。

- 通过字符串创建。
- 通过日期时间的各个成分创建。
- 通过现有的日期时间对象创建。

接下来我们将分别介绍这 3 种方法。

12.2.1 通过字符串创建

日期时间数据经常用字符串来表示。8.3.4 节中已经介绍了将字符串解析为日期时间数据的一种方法。通过字符串创建日期时间数据的另一种方法是使用 lubridate 中提供的辅助函数。如果确定了各个组成部分的顺序，那么这些函数可以自动将字符串转换为日期时间格式。要想使用这些函数，需要先确定年、月和日在日期数据中的顺序，然后按照同样的顺序排列 y、m 和 d，这样就可以组成能够解析日期的 lubridate 函数名称。例如：

```
ymd("2017-01-31")
#> [1] "2017-01-31"
mdy("January 31st, 2017")
#> [1] "2017-01-31"
```

```
dmy("31-Jan-2017")
#> [1] "2017-01-31"
```

这些函数也可以接受不带引号的数值。这是创建单个日期时间对象的最简方法，在筛选日期时间数据时，你就可以使用这种方法。`ymd()` 是个非常简单明了的函数：

```
ymd(20170131)
#> [1] "2017-01-31"
```

`ymd()` 和类似的其他函数可以创建日期。要想创建日期时间型数据，可以在后面加一个下划线，以及 h、m 和 s 之中的一个或多个字母，这样就可以得到解析日期时间的函数了：

```
ymd_hms("2017-01-31 20:11:59")
#> [1] "2017-01-31 20:11:59 UTC"
mdy_hm("01/31/2017 08:01")
#> [1] "2017-01-31 08:01:00 UTC"
```

通过添加一个时区参数，你可以将一个日期强制转换为日期时间：

```
ymd(20170131, tz = "UTC")
#> [1] "2017-01-31 UTC"
```

12.2.2 通过各个成分创建

除了单个字符串，日期时间数据的各个成分还经常分布在表格的多个列中。航班数据就是这样的：

```
flights %>%
  select(year, month, day, hour, minute)
#> # A tibble: 336,776 × 5
#>    year month   day  hour minute
#>   <int> <int> <int> <dbl>  <dbl>
#> 1  2013     1     1     5     15
#> 2  2013     1     1     5     29
#> 3  2013     1     1     5     40
#> 4  2013     1     1     5     45
#> 5  2013     1     1     6      0
#> 6  2013     1     1     5     58
#> # ... with 3.368e+05 more rows
```

如果想要按照这种表示方法来创建日期或时间，可以使用 `make_date()` 函数创建日期，使用 `make_datetime()` 函数创建日期时间：

```
flights %>%
  select(year, month, day, hour, minute) %>%
  mutate(
    departure = make_datetime(year, month, day, hour, minute)
  )
#> # A tibble: 336,776 × 6
#>    year month   day  hour minute           departure
#>   <int> <int> <int> <dbl>  <dbl>              <dttm>
#> 1  2013     1     1     5     15 2013-01-01 05:15:00
#> 2  2013     1     1     5     29 2013-01-01 05:29:00
#> 3  2013     1     1     5     40 2013-01-01 05:40:00
```

```
#> 4  2013     1     1     5     45 2013-01-01 05:45:00
#> 5  2013     1     1     6      0 2013-01-01 06:00:00
#> 6  2013     1     1     5     58 2013-01-01 05:58:00
#> # ... with 3.368e+05 more rows
```

我们对 flights 中的 4 个时间列进行相同的操作。因为时间的表示方法有点奇怪，所以我们先使用模运算将小时成分与分钟成分分离。一旦建立了日期时间变量，我们就在本章剩余部分使用这些变量进行讨论：

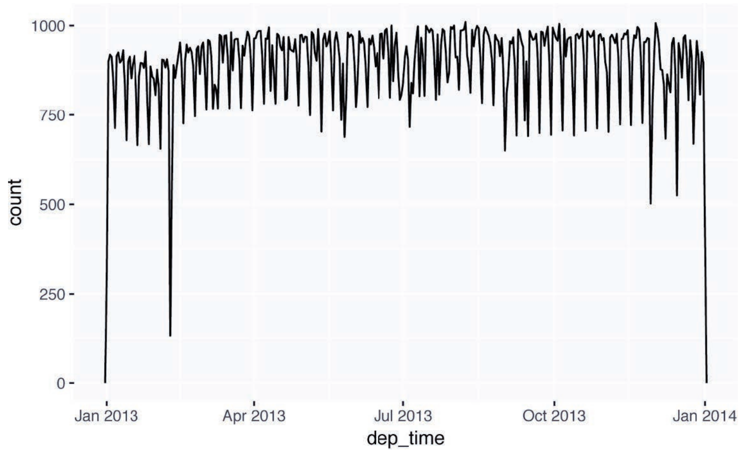
```
make_datetime_100 <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100)
}

flights_dt <- flights %>%
  filter(!is.na(dep_time), !is.na(arr_time)) %>%
  mutate(
    dep_time = make_datetime_100(year, month, day, dep_time),
    arr_time = make_datetime_100(year, month, day, arr_time),
    sched_dep_time = make_datetime_100(
      year, month, day, sched_dep_time
    ),
    sched_arr_time = make_datetime_100(
      year, month, day, sched_arr_time
    )
  ) %>%
  select(origin, dest, ends_with("delay"), ends_with("time"))

flights_dt
#> # A tibble: 328,063 × 9
#>   origin  dest dep_delay arr_delay            dep_time
#>    <chr> <chr>     <dbl>     <dbl>              <dttm>
#> 1    EWR   IAH         2        11 2013-01-01 05:17:00
#> 2    LGA   IAH         4        20 2013-01-01 05:33:00
#> 3    JFK   MIA         2        33 2013-01-01 05:42:00
#> 4    JFK   BQN        -1       -18 2013-01-01 05:44:00
#> 5    LGA   ATL        -6       -25 2013-01-01 05:54:00
#> 6    EWR   ORD        -4        12 2013-01-01 05:54:00
#> # ... with 3.281e+05 more rows, and 4 more variables:
#> #   sched_dep_time <dttm>, arr_time <dttm>,
#> #   sched_arr_time <dttm>, air_time <dbl>
```

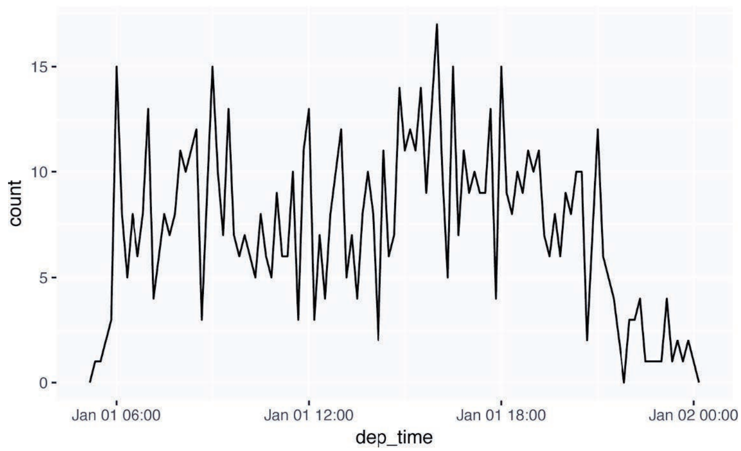
我们可以使用这些数据做出一年间出发时间的可视化分布：

```
flights_dt %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(binwidth = 86400) # 86400秒 = 1天
```



或者一天内的分布：

```
flights_dt %>%
  filter(dep_time < ymd(20130102)) %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(binwidth = 600) # 600秒 = 10分钟
```



注意，当将日期时间型数据作为数值使用时（比如在直方图中），1 表示 1 秒，因此分箱宽度 86 400 才能够表示 1 天。对于日期型数据，1 则表示 1 天。

12.2.3 通过其他类型数据创建

有时你需要在日期时间型数据和日期型数据之间进行转换，这正是 as_datetime() 和 as_date() 函数的功能：

```
as_datetime(today())
#> [1] "2016-10-10 UTC"
as_date(now())
#> [1] "2016-10-10"
```

有时我们会使用“Unix 时间戳”（即 1970-01-01）的偏移量来表示日期时间。如果偏移量单位是秒，那么就使用 as_datetime() 函数来转换；如果偏移量单位是天，则使用 as_date() 函数来转换：

```
as_datetime(60 * 60 * 10)
#> [1] "1970-01-01 10:00:00 UTC"
as_date(365 * 10 + 2)
#> [1] "1980-01-01"
```

12.2.4 练习

(1) 如果解析包含无效日期的一个字符串，那么会发生什么情况？

```
ymd(c("2010-10-10", "bananas"))
```

(2) today() 函数中的 tzone 参数的作用是什么？为什么它很重要？

(3) 使用恰当的 lubridate 函数来解析以下每个日期。

```
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14" # 2014年12月30日
```

12.3 日期时间成分

现在你已经知道了如何将日期时间型数据保存在 R 的相应数据结构中。接下来我们研究一下能够对这些数据进行何种处理。本节将重点介绍获取和设置日期时间成分的访问器函数。下一节将介绍如何对日期时间进行数学运算。

12.3.1 获取成分

如果想要提取出日期中的独立成分，可以使用以下访问器函数：year()、month()、mday()（一个月中的第几天）、yday()（一年中的第几天）、wday()（一周中的第几天）、hour()、minute() 和 second()：

```
datetime <- ymd_hms("2016-07-08 12:34:56")

year(datetime)
#> [1] 2016
month(datetime)
#> [1] 7
mday(datetime)
#> [1] 8

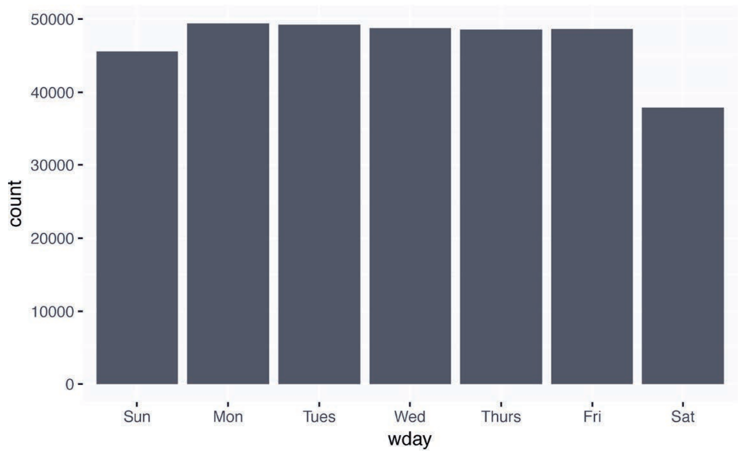
yday(datetime)
#> [1] 190
wday(datetime)
#> [1] 6
```

对于 month() 和 wday() 函数，你可以设置 label = TRUE 来返回月份名称和星期数的缩写，还可以设置 abbr = FALSE 来返回全名：

```
month(datetime, label = TRUE)
#> [1] Jul
#> 12 Levels: Jan < Feb < Mar < Apr < May < Jun < ... < Dec
wday(datetime, label = TRUE, abbr = FALSE)
#> [1] Friday
#> 7 Levels: Sunday < Monday < Tuesday < ... < Saturday
```

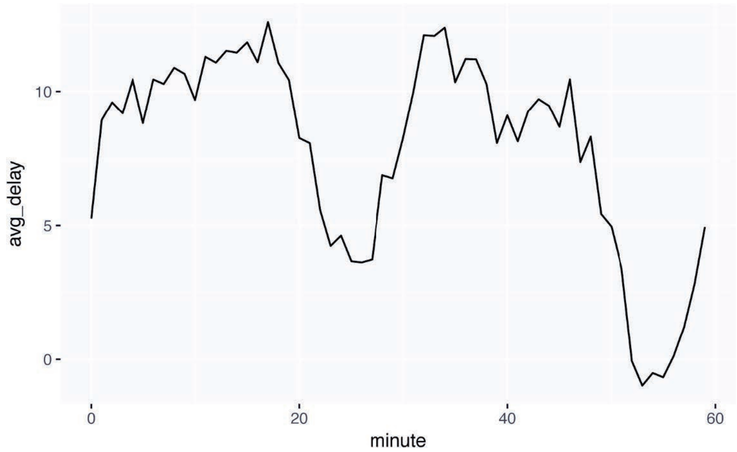
通过 wday() 函数，我们可以知道在工作日出发的航班要多于周末出发的航班：

```
flights_dt %>%
  mutate(wday = wday(dep_time, label = TRUE)) %>%
  ggplot(aes(x = wday)) +
    geom_bar()
```



如果查看一小时内每分钟的平均出发延误，我们可以发现一个有趣的模式。似乎在第 20~30 分钟和第 50~60 分钟内出发的航班的延误时间远远低于其他时间出发的航班！

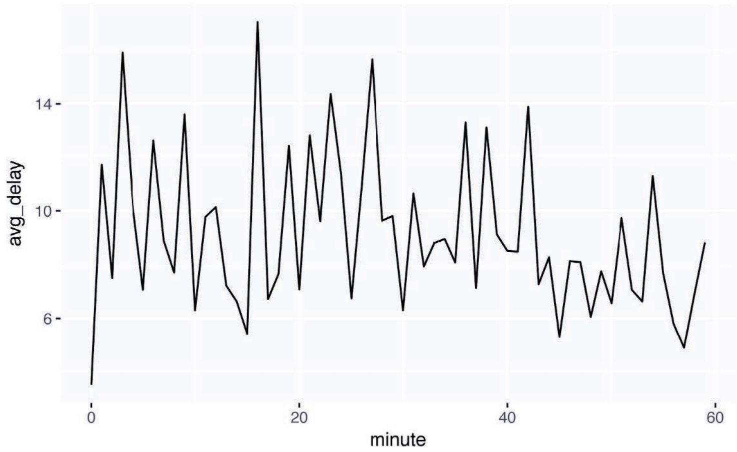
```
flights_dt %>%
  mutate(minute = minute(dep_time)) %>%
  group_by(minute) %>%
  summarize(
    avg_delay = mean(arr_delay, na.rm = TRUE),
    n = n()) %>%
  ggplot(aes(minute, avg_delay)) +
    geom_line()
```



有趣的是，如果检查**计划**出发时间，我们就会发现其中没有这么明显的模式：

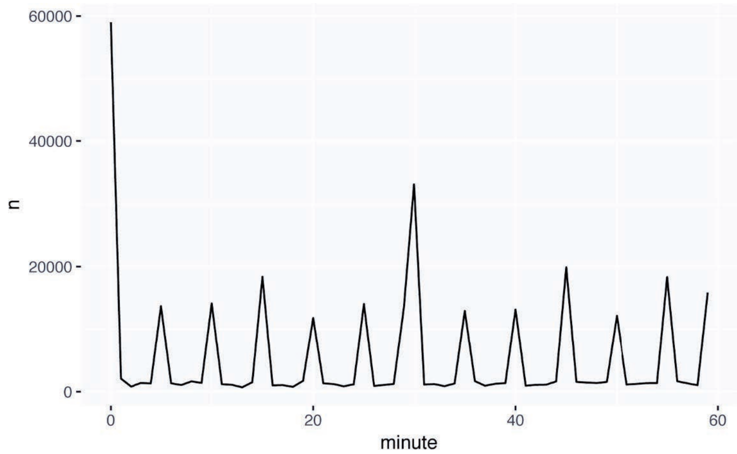
```
sched_dep <- flights_dt %>%
  mutate(minute = minute(sched_dep_time)) %>%
  group_by(minute) %>%
  summarize(
    avg_delay = mean(arr_delay, na.rm = TRUE),
    n = n())

ggplot(sched_dep, aes(minute, avg_delay)) +
  geom_line()
```



那么，为什么我们会在实际的出发时间中看到这种模式呢？好吧，与人工收集的很多数据一样，航班数据也严重倾向于在“美妙的”时间出发的那些航班。只要处理的数据涉及人工判断，那么你就要对这种模式保持警惕！

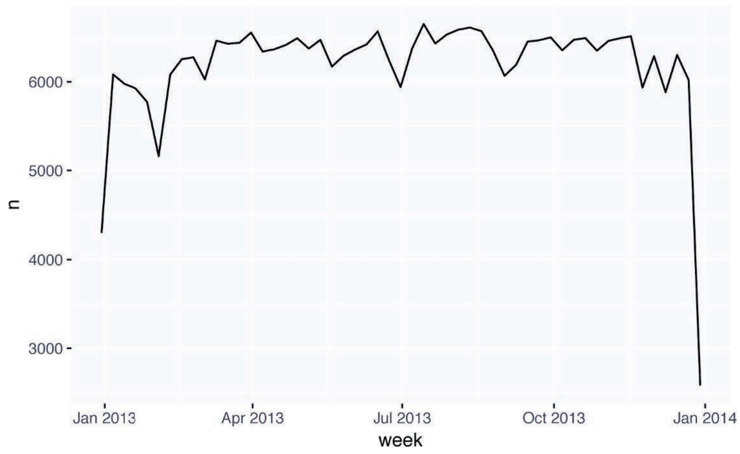
```
ggplot(sched_dep, aes(minute, n)) +
  geom_line()
```



12.3.2　舍入

绘制独立日期成分的另一种方法是，通过 floor_date()、round_date() 和 ceiling_date() 函数将日期舍入到临近的一个时间单位。这些函数的参数都包括一个待调整的时间向量，以及时间单位名称，函数会将这个向量舍下、入上或四舍五入到这个时间单位。例如，以下代码可以绘制出每周的航班数量：

```
flights_dt %>%
  count(week = floor_date(dep_time, "week")) %>%
  ggplot(aes(week, n)) +
    geom_line()
```



计算舍入日期和未舍入日期间的差别是非常有用的。

12.3.3　设置成分

你还可以使用每个访问器函数来设置日期时间中的成分：

```
(datetime <- ymd_hms("2016-07-08 12:34:56"))
#> [1] "2016-07-08 12:34:56 UTC"

year(datetime) <- 2020
datetime
#> [1] "2020-07-08 12:34:56 UTC"
month(datetime) <- 01
datetime
#> [1] "2020-01-08 12:34:56 UTC"
hour(datetime) <- hour(datetime) + 1
```

除了原地修改，你还可以通过 update() 函数创建一个新日期时间。这样也可以同时设置多个成分：

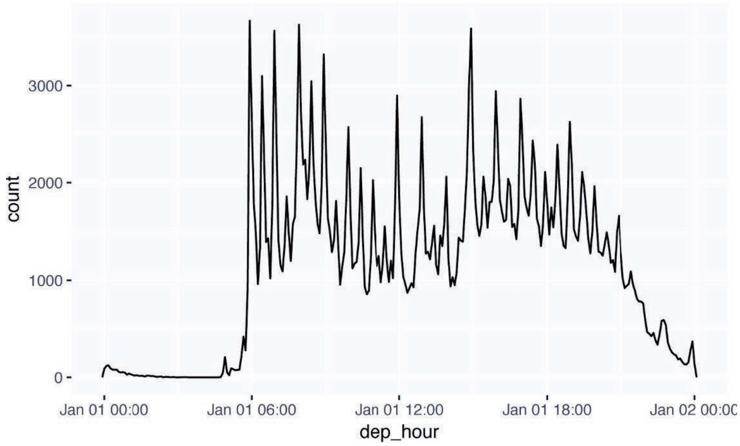
```
update(datetime, year = 2020, month = 2, mday = 2, hour = 2)
#> [1] "2020-02-02 02:34:56 UTC"
```

如果设置的值过大，那么可以自动向后滚动：

```
ymd("2015-02-01") %>%
  update(mday = 30)
#> [1] "2015-03-02"
ymd("2015-02-01") %>%
  update(hour = 400)
#> [1] "2015-02-17 16:00:00 UTC"
```

你可以使用 update() 函数来显示这一年中所有航班的出发时间在一天内的分布：

```
flights_dt %>%
  mutate(dep_hour = update(dep_time, yday = 1)) %>%
  ggplot(aes(dep_hour)) +
    geom_freqpoly(binwidth = 300)
```



将日期中较大的成分设定为常数是探索其中较小成分模式的一种有效手段。

12.3.4 练习

(1) 在一年的范围内，航班时间在一天中的分布是如何变化的？

(2) 比较 dep_time、sched_dep_time 和 dep_delay。它们是一致的吗？解释一下你的结论。

(3) 比较航班出发时间与到达时间之间的间隔和 air_time。解释一下你的结论。（提示：考虑一下机场所在地点。）

(4) 平均延误时间在一天范围内是如何变化的？应该使用 dep_time 还是 sched_dep_time？为什么？

(5) 如果想要将延误的几率降至最低，那么应该在星期几搭乘航班？

(6) 什么因素使得 diamonds$carat 和 flights$sched_dep_time 的分布很相似？

(7) 确认假设：在第 20~30 分钟和第 50~60 分钟内，航班能提前起飞的原因是其他航班提前起飞了。提示：创建一个二值变量来表明航班是否有延误。

12.4 时间间隔

接下来你将学习如何对日期进行数学运算，其中包括减法、加法和除法。本节将介绍 3 种用于表示时间间隔的重要类。

- **时期**：以秒为单位表示一段精确的时间。
- **阶段**：表示由人工定义的一段时间，如几周或几个月。
- **区间**：表示从起点到终点的一段时间。

12.4.1 时期

在 R 中，如果将两个日期相减，那么你会得到不同的对象：

```
# Hadley多大了？
h_age <- today() - ymd(19791014)
h_age
#> Time difference of 13511 days
```

表示时间差别的对象记录时间间隔的单位可以是秒、分钟、小时、日或周。因为这种模棱两可的对象处理起来非常困难，所以 lubridate 提供了总是使用秒为单位的另一种计时对象——**时期**：

```
as.duration(h_age)
#> [1] "1167350400s (~36.99 years)"
```

可以使用很多方便的构造函数来创建时期：

```
dseconds(15)
#> [1] "15s"
dminutes(10)
#> [1] "600s (~10 minutes)"
dhours(c(12, 24))
```

```
#> [1] "43200s (~12 hours)" "86400s (~1 days)"
ddays(0:5)
#> [1] "0s"                "86400s (~1 days)"
#> [3] "172800s (~2 days)" "259200s (~3 days)"
#> [5] "345600s (~4 days)" "432000s (~5 days)"
dweeks(3)
#> [1] "1814400s (~3 weeks)"
dyears(1)
#> [1] "31536000s (~52.14 weeks)"
```

时期总是以秒为单位来记录时间间隔。使用标准比率（1 分钟为 60 秒、1 小时为 60 分钟、1 天为 24 小时、1 周为 7 天、1 年为 365 天）将分钟、小时、日、周和年转换为秒，从而建立具有更大值的对象。

可以对时期进行加法和乘法操作：

```
2 * dyears(1)
#> [1] "63072000s (~2 years)"
dyears(1) + dweeks(12) + dhours(15)
#> [1] "38847600s (~1.23 years)"
```

时期可以和日期型数据相加或相减：

```
tomorrow <- today() + ddays(1)
last_year <- today() - dyears(1)
```

然而，因为时期表示的是以秒为单位的一段精确时间，所以有时你会得到意想不到的结果：

```
one_pm <- ymd_hms(
  "2016-03-12 13:00:00",
  tz = "America/New_York"
)

one_pm
#> [1] "2016-03-12 13:00:00 EST"
one_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
```

为什么 3 月 12 日下午 1 点的 1 天后变成了 3 月 13 日的下午 2 点？如果仔细观察，你就会发现时区发生了变化。因为夏时制，3 月 12 日只有 23 个小时，因此如果我们加上一整天的秒数，那么就会得到一个不正确的时间。

12.4.2 阶段

为了解决这个问题，lubridate 提供了**阶段**对象。阶段也是一种时间间隔，但它不以秒为单位；相反，它使用“人工”时间，比如日和月。这使得它们使用起来更加直观：

```
one_pm
#> [1] "2016-03-12 13:00:00 EST"
one_pm + days(1)
#> [1] "2016-03-13 13:00:00 EDT"
```

和时期一样，我们也可以使用很多友好的构造函数来创建阶段：

```
seconds(15)
#> [1] "15S"
minutes(10)
#> [1] "10M 0S"
hours(c(12, 24))
#> [1] "12H 0M 0S" "24H 0M 0S"
days(7)
#> [1] "7d 0H 0M 0S"
months(1:6)
#> [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S"
#> [4] "4m 0d 0H 0M 0S" "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
weeks(3)
#> [1] "21d 0H 0M 0S"
years(1)
#> [1] "1y 0m 0d 0H 0M 0S"
```

可以对阶段进行加法和乘法操作：

```
10 * (months(6) + days(1))
#> [1] "60m 10d 0H 0M 0S"
days(50) + hours(25) + minutes(2)
#> [1] "50d 25H 2M 0S"
```

当然，阶段可以和日期相加。与时期相比，阶段更容易符合我们的预期：

```
# 闰年
ymd("2016-01-01") + dyears(1)
#> [1] "2016-12-31"
ymd("2016-01-01") + years(1)
#> [1] "2017-01-01"

# 夏时制
one_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
one_pm + days(1)
#> [1] "2016-03-13 13:00:00 EDT"
```

下面我们使用阶段来解决与航班日期有关的一个怪现象。有些飞机似乎在从纽约市起飞**前**就到达了目的地：

```
flights_dt %>%
  filter(arr_time < dep_time)
#> # A tibble: 10,633 × 9
#>   origin  dest dep_delay arr_delay            dep_time
#>    <chr> <chr>     <dbl>     <dbl>              <dttm>
#> 1    EWR   BQN         9        -4 2013-01-01 19:29:00
#> 2    JFK   DFW        59        NA 2013-01-01 19:39:00
#> 3    EWR   TPA        -2         9 2013-01-01 20:58:00
#> 4    EWR   SJU        -6       -12 2013-01-01 21:02:00
#> 5    EWR   SFO        11       -14 2013-01-01 21:08:00
#> 6    LGA   FLL       -10        -2 2013-01-01 21:20:00
#> # ... with 1.063e+04 more rows, and 4 more variables:
#> #   sched_dep_time <dttm>, arr_time <dttm>,
#> #   sched_arr_time <dttm>, air_time <dbl>
```

这些都是过夜航班。我们使用了同一种日期来表示出发时间和到达时间，但这些航班是在第二天到达的。将每个过夜航班的到达时间加上一个 days(1)，就可以解决这个问题了：

```
flights_dt <- flights_dt %>%
  mutate(
    overnight = arr_time < dep_time,
    arr_time = arr_time + days(overnight * 1),
    sched_arr_time = sched_arr_time + days(overnight * 1)
  )
```

这样一来，航班数据就符合常理了：

```
flights_dt %>%
  filter(overnight, arr_time < dep_time)
#> # A tibble: 0 × 10
#> # ... with 10 variables: origin <chr>, dest <chr>,
#> #   dep_delay <dbl>, arr_delay <dbl>, dep_time <dttm>,
#> #   sched_dep_time <dttm>, arr_time <dttm>,
#> #   sched_arr_time <dttm>, air_time <dbl>, overnight <lgl>
```

12.4.3　区间

显然，dyears(1) / ddays(365) 应该返回 1，因为时期总是以秒来表示的，表示 1 年的时期就定义为相当于 365 天的秒数。

那么 years(1) / days(1) 应该返回什么呢？如果年份是 2015 年，那么结果就是 365，但如果年份是 2016 年，那么结果就是 366！没有足够的信息让 lubridate 返回一个明确的结果。lubridate 的做法是给出一个估计值，同时给出一条警告：

```
years(1) / days(1)
#> estimate only: convert to intervals for accuracy
#> [1] 365
```

如果需要更精确的测量方式，那么你就必须使用**区间**。区间是带有起点的时期，这使得其非常精确，你可以确切地知道它的长度：

```
next_year <- today() + years(1)
(today() %--% next_year) / ddays(1)
#> [1] 365
```

要想知道一个区间内有多少个阶段，你需要使用整数除法：

```
(today() %--% next_year) %/% days(1)
#> [1] 365
```

12.4.4　小结

如何在时期、阶段和区间中进行选择呢？一如既往，选择能够解决问题的最简单的数据结构。如果只关心物理时间，那么就使用时期；如果还需要考虑人工时间，那么就使用阶段；如果需要找出人工时间范围内有多长的时间间隔，那么就使用区间。

图 12-1 总结了不同数据类型之间可以进行的数学运算。

	日期	日期时间	时期	阶段	区间	数值
日期	-		- +	- +		- +
日期时间		-	- +	- +		- +
时期	- +	- +	- + /			- + × /
阶段	- +	- +		- +		- + × /
区间			/	/		
数值	- +	- +	- + ×	- + ×	- + ×	- + × /

图 12-1：不同日期 / 时间类之间可以进行的数学运算

12.4.5 练习

(1) 为什么存在 `months()` 函数，但没有 `dmonths()` 函数？

(2) 向刚开始学习 R 的人解释 `days(overnight * 1)` 的意义。它的作用是什么？

(3) 创建一个日期向量来给出 2015 年每个月的第一天；创建一个日期向量来给出**今年**每个月的第一天。

(4) 编写一个函数，输入你的生日（日期型），使其返回你的年龄（以年为单位）。

(5) 为什么 `(today() %--% (today() + years(1)) / months(1)` 这段代码不能正常运行？

12.5 时区

时区是个非常复杂的主题，因为其受很多地理政治因素的影响。好在我们无须对其细节进行过多研究，因为它对数据分析来说不是特别重要，但还是有一些问题需要我们解决。

第一个问题是，时区的名称有些含糊不清。例如，如果你是美国人，那么一定很熟悉 EST（Eastern Standard Time，东部标准时间）。但是，澳大利亚和加拿大也都有 EST！ 为了避免混淆，R 使用国际标准 IANA 时区。这些时区使用统一带有“/”的命名方式，一般的形式为“< 大陆 >/< 城市 >”（存在例外，因为不是所有城市都位于一块大陆）。如“America/New_York”、“Europe/Paris”和“Pacific/Auckland”。

你可能很想知道，为什么时区使用城市来表示，通常我们都会认为时区应该使用国家或地区来表示。这是因为 IANA 数据库必须记录十年间的时区，而在十年时间中，国家更名（或分裂）的情况非常多，但城市名是基本保持不变的。另一个问题是，时区名称要反映的不仅是现在的情况，还有整个历史。例如，有两个名为“America/New_York”和“America/Detroit”的时区。这两个城市现在都使用东部标准时间，但在 1969~1972 年，密歇根（底特律所在的州）不使用夏时制，因此它需要一个不同的时区名称。仅仅是这些故事，就值得你阅读一下原始时区数据库！

在 R 中，可以使用 `Sys.timezone()` 函数找出你的当前时区：

```
Sys.timezone()
#> [1] "America/Los_Angeles"
```

（如果 R 找不到时区，那么就会返回 NA。）

你还可以使用 OlsonNames() 函数来查看完整的时区名称列表：

```
length(OlsonNames())
#> [1] 589
head(OlsonNames())
#> [1] "Africa/Abidjan"     "Africa/Accra"
#> [3] "Africa/Addis_Ababa" "Africa/Algiers"
#> [5] "Africa/Asmara"      "Africa/Asmera"
```

在 R 中，时区是日期时间型数据的一个属性，仅用于控制输出。例如，以下 3 个对象表示的是同一时刻：

```
(x1 <- ymd_hms("2015-06-01 12:00:00", tz = "America/New_York"))
#> [1] "2015-06-01 12:00:00 EDT"
(x2 <- ymd_hms("2015-06-01 18:00:00", tz = "Europe/Copenhagen"))
#> [1] "2015-06-01 18:00:00 CEST"
(x3 <- ymd_hms("2015-06-02 04:00:00", tz = "Pacific/Auckland"))
#> [1] "2015-06-02 04:00:00 NZST"
```

你可以使用减法操作来验证它们都是同一时刻：

```
x1 - x2
#> Time difference of 0 secs
x1 - x3
#> Time difference of 0 secs
```

除非使用了其他设置，lubridate 总是使用 UTC（Coordinated Universal Time，国际标准时间）。UTC 是科技界使用的时区标准，基本等价于它的前身 GMT（Greenwich Mean Time，格林尼治标准时间）。因为没有夏时制，所以它非常适合计算。涉及日期时间的操作（比如 c()）经常会丢弃时区信息。在这种情况下，日期时间会显示你的本地时区：

```
x4 <- c(x1, x2, x3)
x4
#> [1] "2015-06-01 09:00:00 PDT" "2015-06-01 09:00:00 PDT"
#> [3] "2015-06-01 09:00:00 PDT"
```

你可以使用两种方法来改变时区。

- 保持时间不变，修改其显示方式。当时间正确，但需要更直观的表示时，可以使用这种方法：

```
x4a <- with_tz(x4, tzone = "Australia/Lord_Howe")
x4a
#> [1] "2015-06-02 02:30:00 LHST"
#> [2] "2015-06-02 02:30:00 LHST"
#> [3] "2015-06-02 02:30:00 LHST"
x4a - x4
#> Time differences in secs
#> [1] 0 0 0
```

（这同时反映出时区的另一个问题：它们的偏移小时数不全是整数！）

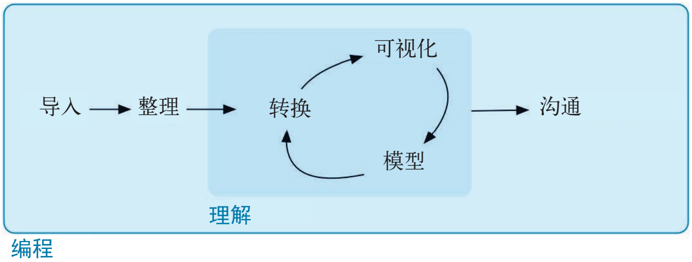
- 修改内部时间。当时间数据被标注了错误的时区，而想要改正过来时，你就可以使用这种方法：

```
x4b <- force_tz(x4, tzone = "Australia/Lord_Howe")
x4b
#> [1] "2015-06-01 09:00:00 LHST"
#> [2] "2015-06-01 09:00:00 LHST"
#> [3] "2015-06-01 09:00:00 LHST"
x4b - x4
#> Time differences in hours
#> [1] -17.5 -17.5 -17.5
```

第三部分

编程

这一部分致力于提高你的编程技能。编程是所有数据科学工作都不可或缺的一项技能：你必须使用计算机来进行数据科学工作，不可能在头脑中或使用纸和笔来进行。



编程产出代码，代码是一种沟通工具。很显然，代码可以告诉计算机你想要做什么，同时也可以用于人与人之间的交流。将代码当作一种沟通工具是非常重要的，因为现在所有项目基本上都要靠协作才能完成。即使你现在单枪匹马地工作，也肯定要与未来的自己进行交流！代码清晰易懂特别重要，这样其他人（包括未来的你）才能理解你为什么要使用这种方式进行分析。因此，提高编程能力的同时也要提高沟通能力。随着时间的推移，你不仅会希望代码更易于编写，还会希望它更容易为他人所理解。

写代码和写文章在很多方面是非常相似的。我们发现二者特别重要的一个共同之处是，要想让代码或文章更加清晰易懂，关键是重写。你对自己想法的第一次表达往往不是特别清晰，因此需要多次重写。解决一个数据分析难题后，我们通常应该再审视一下代码，思考一下它是否真正实现了我们的要求。当产生新想法时，如果花一点时间重写代码，就可以节省以后重构代码所需的大量时间。但这并不是说你应该重写所有功能。是尽快实现当前功能，还是从长远来看节约时间，你需要权衡一下。（不过，功能重写得越多，就越可能将最初想法表达得更清楚。）

以下 4 章中介绍的技能既可以帮助你编写新的程序，又能够让你更清晰、更容易地解决现有问题。

- 第 13 章将深入介绍**管道操作**，即 `%>%`，你将学习更多关于管道操作的工作原理和替代方式，以及不适合使用管道的情形。
- 复制粘贴确实功能强大，但这种操作不应该超过两次。代码中的重复内容是非常危险的，因为这样很容易导致错误和不一致。第 14 章会介绍如何编写**函数**，这是重复使用代码的一种方式，它可以让你提取出重复代码，然后轻松地进行重用。
- 当开始编写功能更强大的函数时，你需要深刻理解 R 的**数据结构**，这就是第 15 章的内容。你必须掌握 4 种常用的原子向量，以及以此为基础构建的 3 种重要 S3 类，并理解列表和数据框背后的奥秘。
- 函数可以提取出重复代码，但你经常需要对不同的输入重复相同的操作。你需要可以多次执行相同操作的**迭代**工具，这些工具包括 `for` 循环和函数式编程，这就是第 16 章将要介绍的内容。

更多学习资源

以上各章的目的是让你掌握实践数据科学所必需的编程技能，其中的内容还是相当多的。如果你已经完全掌握了本书中的内容，我们坚信你还应该进一步拓展你的编程技能。学习更多编程技能是一项长期投资，虽然其效果不是立竿见影，但从长期来看，它可以让你更高效地解决新问题，也可以让你将从以前问题中获得的知识和经验应用于新的场景。

为了学到更多知识，你需要将 R 当作一门编程语言，而不只是数据科学的一种交互环境。我们已经出版了两本书来帮助你学习 R 编程。

- 《R 语言入门与实践》，Garrett Grolemund 著。这是 R 编程语言的一本入门书，如果 R 是你的第一门编程语言，那么从该书开始是非常合适的。该书的内容与上述各章非常相似，但使用了不同的风格和示例。如果你觉得这 4 章的内容过于简略，那么该书是一项重要补充。
- *Advanced R*，Hadley Wickham 著。该书深入介绍了 R 作为编程语言的各种细节。如果你已有编程经验，那么该书非常适合你。如果你已经掌握了上述 4 章的内容，那么该书也非常适合你继续学习。该书有在线版本。

第13章

使用magrittr进行管道操作

13.1 简介

管道是一种强大的工具，可以清楚地表示由多个操作组成的一个操作序列。到目前为止，我们已经知道了如何使用管道，但还不清楚其工作原理，也不知道它是否有替代方式。本章将更加详细地研究管道操作。你将学到管道的替代方式、何时不应该使用管道，以及其他一些有用的相关工具。

准备工作

管道 `%>%` 来自于 Stefan Milton Bache 开发的 magrittr 包。因为 tidyverse 中的包会自动加载 `%>%`，所以通常你无须显式地加载 magrittr。但接下来我们将重点讨论管道操作，且不加载任何其他 R 包，因此需要显式地加载 magrittr 包。

```
library(magrittr)
```

13.2 管道的替代方式

管道操作的出发点是帮助你以清晰易懂的方式编写代码。为了说明管道如此有用的原因，我们将探究同一段代码的不同编写方式。现在我们使用代码来讲述小兔福福的故事：

> 一只小兔叫福福
> 蹦蹦跳跳过森林
> 抓起一窝小田鼠
> 每只头上打一下

这是一首流传甚广的童谣，可以边唱边配合手部动作。

首先，我们定义一个对象来表示小兔福福：

```
foo_foo <- little_bunny()
```

然后，我们使用函数来表示每个动作：hop()、scoop() 和 bop()。通过这个对象和这些函数，我们至少有 4 种方法来使用代码讲述这个故事：

- 将每个中间步骤保存为一个新对象；
- 多次重写初始对象；
- 组合多个函数；
- 使用管道。

接下来我们会依次介绍每种方法，并讨论其优缺点。

13.2.1 中间步骤

最简单的方法是将每个中间步骤保存为一个新对象：

```
foo_foo_1 <- hop(foo_foo, through = forest)
foo_foo_2 <- scoop(foo_foo_1, up = field_mice)
foo_foo_3 <- bop(foo_foo_2, on = head)
```

这种方法的最大缺点是，你必须为每个中间结果建立一个变量。如果这些变量确实有意义，那么这就是一种好方法，你也应该建立这些变量。但在很多情况下，比如在以上示例中，这些变量其实是没有什么实际意义的，你还必须使用数字后缀来区分这些变量。这样会造成两个问题。

- 代码中充斥着大量不必要的变量。
- 你必须在每一行代码中小心翼翼地修改变量后缀。

每次编写这种代码，我都会在某行用错数字，然后花 10 分钟一边挠着脑袋，一边努力找出代码的错误。

你可能还会担心这种代码创建的多个数据副本会占用大量内存。出人意料的是，这种担心大可不必。首先，你不应该将时间花在过早担心内存上。当内存确实成为问题（即内存耗尽）时再担心便是，否则就是杞人忧天。其次，R 是很智能的，它会尽量在数据框之间共享数据列。以一个实际的数据处理流程为例，我们向数据集 ggplot2::diamonds 中添加一个新列：

```
diamonds <- ggplot2::diamonds
diamonds2 <- diamonds %>%
  dplyr::mutate(price_per_carat = price / carat)

pryr::object_size(diamonds)
#> 3.46 MB
pryr::object_size(diamonds2)
#> 3.89 MB
pryr::object_size(diamonds, diamonds2)
#> 3.89 MB
```

函数 pryr::object_size() 会返回其所有参数占用的内存。以下结果乍一看似乎有悖常理。

- diamonds 占用了 3.46MB 内存。
- diamonds2 占用了 3.89MB 内存。
- diamonds 和 diamonds2 一共占用了 3.89MB 内存！

怎么会这样呢？ diamonds2 和 diamonds 有 10 个公共的数据列，这些列中的数据没必要再复制一份，因此这两个数据框中有公用变量。公用变量只有在修改时才会进行复制。在以下的示例中，我们修改了 diamonds$carat 中的一个值。这意味着 carat 变量不再由两个数据框共享，而必须创建一个副本。每个数据框的大小保持不变，但总的大小增加了：

```
diamonds$carat[1] <- NA
pryr::object_size(diamonds)
#> 3.46 MB
pryr::object_size(diamonds2)
#> 3.89 MB
pryr::object_size(diamonds, diamonds2)
#> 4.32 MB
```

（注意，我们在该示例中使用了 pryr::object_size()，而不是内置函数 object_size。因为 object_size() 只能接受一个参数，所以它无法计算在多个对象间共享的数据所占用的空间。）

13.2.2 重写初始对象

除了为每个中间步骤创建新对象，我们还可以重写初始对象：

```
foo_foo <- hop(foo_foo, through = forest)
foo_foo <- scoop(foo_foo, up = field_mice)
foo_foo <- bop(foo_foo, on = head)
```

因为这种方式需要的输入更少（也无须过多思考），所以不容易出错，但是有两个问题。

- 调试起来太痛苦了。如果出错，那么你就必须从头开始运行整个流程。
- 对象的多次重写（输入了 6 次 foo_foo！）阻碍我们看清每行代码中发生的变化。

13.2.3 函数组合

另一种方法是将多个函数组合在一起，这样可以避免赋值语句：

```
bop(
  scoop(
    hop(foo_foo, through = forest),
    up = field_mice
  ),
  on = head
)
```

这种方法的缺点是，必须按照从内向外和从右向左的顺序阅读代码，而且参数太分散了（人们形象地将这种代码称为多层三明治）。简而言之，这种代码不适合人类阅读。

13.2.4 使用管道

最后，我们可以使用管道：

```
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mouse) %>%
  bop(on = head)
```

这种方式是我们的最爱，因为它的重点在于动词，而不是名词。在阅读这一串函数组合时，你可以将它们当成一系列规定动作。福福蹦跳着，然后抓田鼠，接着打田鼠。至于这种方式的缺点，当然就是你必须先熟悉管道。如果以前从来没见过 `%>%`，那么你根本搞不清这段代码的意义。幸好大多数人都可以很快理解这种思想，因此，当与不熟悉管道操作的人分享这种代码时，你可以很轻松地教会他们。

管道的工作原理就是进行“词法变换”。在这种方式背后，magrittr 会重新组合管道代码，按照重写中间变量的方式来运行。当执行以上示例中的管道操作时，实际上 magrittr 执行的是类似以下的代码：

```
my_pipe <- function(.) {
  . <- hop(., through = forest)
  . <- scoop(., up = field_mice)
  bop(., on = head)
}
my_pipe(foo_foo)
```

这意味着管道不能支持以下两类函数。

- 使用当前环境的函数。例如，`assign()` 函数会在当前环境中使用给定名称创建一个新变量：

  ```
  assign("x", 10)
  x
  #> [1] 10

  "x" %>% assign(100)
  x
  #> [1] 10
  ```

  通过管道方式使用 `assign()` 函数是无效的，因为这时赋值操作是在由 `%>%` 建立的一个临时环境中进行的。如果要通过管道方式来使用 `assign()`，就必须显式地指定环境：

  ```
  env <- environment()
  "x" %>% assign(100, envir = env)
  x
  #> [1] 100
  ```

  具有这个问题的其他函数包括 `get()` 和 `load()`。

- 使用惰性求值的函数。在 R 中，不会在函数调用前计算这种函数的参数，只在函数使用时才进行计算。管道依次计算每个参数，因此不能用在这种函数上。

具有这种问题的一个函数是 tryCatch()，它可以捕获并处理程序错误：

```
tryCatch(stop("!"), error = function(e) "An error")
#> [1] "An error"

stop("!") %>%
  tryCatch(error = function(e) "An error")
#> Error in eval(expr, envir, enclos): !
```

使用惰性求值的函数还是很多的，其中包括 R 基础包中的 try()、suppressMessages() 和 suppressWarnings()。

13.3　不适合使用管道的情形

管道是一种功能强大的工具，但并不是你的唯一选择，也不是"万能药"。管道最大的用武之地是重写一段较短的线性操作序列。对于以下几种情形，我们认为最好不要使用管道。

- 操作步骤超过 10（参考值）个。这种情况下，应该使用有意义的变量来保存中间结果。这样会使得调试更加容易，因为你更容易检查中间结果；还可以使得代码更容易理解，因为有意义的变量名称可以帮助别人明白你的代码意图。
- 有多个输入和输出。如果需要处理的不是一个基本对象，而是组合在一起的两个或多个对象，就不要使用管道。
- 操作步骤构成一张具有复杂依赖关系的有向图。管道基本上是一种线性操作，如果使用它来表示复杂的关系，通常会使得代码混乱不清。

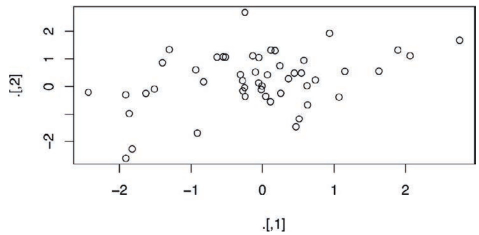
13.4　magrittr中的其他工具

tidyverse 中的所有包都会自动加载 %>%，因此一般不用显式加载 magrittr。然而，magrittr 包中还有其他一些有用的工具，你或许想要尝试一下。

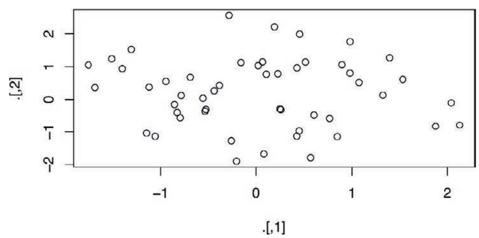
- 在使用比较复杂的管道操作时，有时会因为某个函数的副作用而调用它。比如，你可能想要打印或绘制出当前对象，或者想将它保存在硬盘中。很多时候这种函数不会返回任何结果，只会有效地结束管道操作。

  为了解决这个问题，你可以使用"T"管道操作符 %T>%。它的用法和 %>% 差不多，只是它返回的是左侧项而不是右侧项。之所以称它为"T"操作符，是因为它起的作用类似于 T 形三通管道：

  ```
  rnorm(100) %>%
    matrix(ncol = 2) %>%
    plot() %>%
    str()
  #>  NULL
  ```




```
rnorm(100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()
#>  num [1:50, 1:2] -0.387 -0.785 -1.057 -0.796 -1.756 ...
```



- 如果使用的函数不是基于数据框的（也就是说，你必须传给这些函数一个独立的向量，不能传给它们数据框或基于数据框求值的表达式），那么你就会发现爆炸操作符 %$% 的妙处。它可以将数据框中的变量“炸出来”，让你显式地引用。当需要使用 R 基础包中的很多函数时，这个操作符特别奏效：

```
mtcars %$%
  cor(disp, mpg)
#> [1] -0.848
```

- magrittr 提供了 %<>% 操作符来执行赋值操作，它可以将以下代码：

```
mtcars <- mtcars %>%
  transform(cyl = cyl * 2)
```

  替代为

```
mtcars %<>% transform(cyl = cyl * 2)
```

  我不是很喜欢这个操作符，因为我认为赋值是一种非常特殊的操作，如果需要进行赋值，那么就应该使赋值语句尽量清晰。我的看法是，一点小小的重复（即重复输入对象名称两次）是必要的，它可以更加明确地表示出赋值语句。

第 14 章
函数

14.1 简介

要想成为一名优秀的数据科学家，编写函数绝对是最好的途径之一。与复制粘贴相比，函数可以通过更强大、更通用的方式来自动执行常用任务。相比于复制粘贴，函数具有以下 3 个主要优点。

- 你可以给函数起一个意味深长的名字，从而让代码更容易理解。
- 如果需求发生了变化，只需要修改一处代码即可，无须修改多处。
- 消除了复制粘贴时可能出现的无心之失（比如，修改了一处的变量名称，但却没有修改另一处）。

编写优秀函数可以作为一个人的毕生事业。即使已经使用 R 多年，但我们还是对新技术和使用更好的方法解决老问题孜孜以求。本章的目的不是向你传授编写函数的神功秘籍，而是提供一些行之有效的实用建议，让你能够尽快开始编写自己的函数。

除了函数开发，本章还会对代码风格提出一些建议。良好的代码风格就像正确使用标点符号一样重要。不使用标点符号你也一样可以写文章，但使用标点符号肯定可以让文章更通俗易懂。代码风格种类繁多，各具特色。这里介绍的只是我们所使用的代码风格，最重要的是风格要保持一致。

准备工作

本章的焦点是在 R 基础包中编写函数，因此不需要任何额外的包。

14.2 什么时候应该使用函数

只要一段代码需要复制粘贴的次数超过两次（也就是说，同一段代码至少有 3 个副本），那么就应该考虑编写一个函数。例如，查看下面这段代码，它的功能是什么呢？

```
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

你可能已经看出来了，这段代码的作用是将每列的值调整到 0 到 1 之间。但你能否发现其中的错误呢？在复制粘贴处理 `df$b` 的代码时，我们犯了一个错误：我们忘记将其中的一个 `a` 改成 `b` 了。提取重复代码，将其转换为函数是一种非常好的做法，因为这样可以防止此类错误的发生。

要想编写一个函数，首先需要分析代码，比如，函数需要多少个输入？

```
(df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
```

这段代码只有一个输入：`df$a`。（如果你很好奇为什么 `TRUE` 不是输入，那么可以在后面的练习中找到答案。）为了让输入更加清晰，应该使用具有通用名称的临时变量来重写代码。以上代码只需要一个数值向量，我们可以称其为 `x`：

```
x <- df$a
(x - min(x, na.rm = TRUE)) /
(max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
#>  [1] 0.289 0.751 0.000 0.678 0.853 1.000 0.172 0.611 0.612
#>  [10] 0.601
```

这段代码中还有一些重复，我们计算了 3 次数据最大值和最小值，但这可以一步完成：

```
rng <- range(x, na.rm = TRUE)
(x - rng[1]) / (rng[2] - rng[1])
#>  [1] 0.289 0.751 0.000 0.678 0.853 1.000 0.172 0.611 0.612
#>  [10] 0.601
```

将中间计算结果保存为命名变量是一种非常好的做法，因为这样可以让代码的意义更加清楚。既然我们已经简化了代码，并检验了代码可以正常运行，接下来就可以将其转换为函数了：

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0, 5, 10))
#> [1] 0.0 0.5 1.0
```

要想创建一个新函数，需要以下 3 个关键步骤。

(1) 为函数选择一个**名称**。在以上示例中，我们使用 `rescale01` 作为函数名称，因为这个函数的功能是将一个向量调整到 0 到 1 之间。

(2) 列举出 `function` 中所用的输入，即参数。这个示例中只有一个参数，如果有更多参数，那么函数调用形式就类似于 `function(x, y, z)`。

(3) 将已经编写好的代码放在**函数体**中。在 `function(...)` 后面要紧跟一个用 `{}` 括起来的代码块。

注意以上创建函数的整体过程。确定函数如何使用简单输入来运行后，我们才开始编写函数。从工作代码开始，再将其转换为函数是相对容易的；先创建函数，再让其正确运行则是比较困难的。

此时我们应该使用其他输入来测试函数是否正确：

```
rescale01(c(-10, 0, 10))
#> [1] 0.0 0.5 1.0
rescale01(c(1, 2, 3, NA, 5))
#> [1] 0.00 0.25 0.50   NA 1.00
```

编写了越来越多的函数后，你最终会想要将这种非正式的交互测试过程转化为一种正式的自动化测试过程。这种过程称为单元测试。遗憾的是，它超出了本书的讨论范围，如果想要了解更多关于单元测试的知识，可以访问 http://r-pkgs.had.co.nz/tests.html。

既然已经有了函数，那么我们就可以利用它来简化原来的示例了：

```
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

相对于原来的代码，这段代码更清楚易懂，而且还消除了复制粘贴可能带来的错误。但这段代码中仍然有一些重复，因为我们对多个数据列进行了同样的操作。你将在第 16 章中学习如何消除这种重复，前提是你在第 15 章中学习了更多关于 R 数据结构的知识。

函数的另一个优点是，如果需求发生变化，我们只需要在一处进行修改。例如，我们发现，如果有些变量中包括无限值，那么 `rescale01()` 函数就会出错：

```
x <- c(1:10, Inf)
rescale01(x)
#>  [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

因为已经将代码放在函数中了，所以我们只需要在函数中进行修改即可：

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(x)
#>  [1] 0.000 0.111 0.222 0.333 0.444 0.556 0.667 0.778 0.889
#>  [10] 1.000   Inf
```

这个示例非常好地体现了“不要重复自己”（do not repeat yourself，DRY）这一原则。代码中的重复部分越多，当事情发生变化时（这是必然的），你需要修改的地方就越多，随着时间的推移，代码中的隐患也会越来越多。

练习

(1) 为什么 `TRUE` 不是 `rescale01()` 函数的参数？如果 `x` 中包含一个缺失值，而且 `na.rm` 的值是 `FALSE`，那么会发生什么情况？

(2) 在 `rescale01()` 函数的第二个版本中，无穷大值未作任何处理。重写 `rescale01()` 函数，将 `-Inf` 映射为 0，`Inf` 映射为 1。

(3) 将下面的代码片段转换成函数。思考一下每个函数的作用，你应该为新函数选择什么样的名称？它需要几个参数？能否重写代码，让函数更清晰易懂，并减少重复的操作？

```
mean(is.na(x))

x / sum(x, na.rm = TRUE)

sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)
```

(4) 参考 http://nicercode.github.io/intro/writing-functions.html，编写一个函数来计算数值向量的方差和偏度。

(5) 编写一个名为 `both_na()` 的函数，它接受两个长度相同的向量，如果两个向量中同一位置的值都是 `NA`，则返回这样位置的数量。

(6) 以下函数的功能是什么？尽管它们都很简短，但是用处都很大，为什么？

```
is_directory <- function(x) file.info(x)$isdir
is_readable <- function(x) file.access(x, 4) == 0
```

(7) 阅读“小兔福福”的完整歌词（可以查看维基百科上的 Little Bunny Foo Foo）。这首歌中有大量重复内容。扩展原来的管道操作示例来重新生成完整的歌谣，并使用函数来减少重复。

14.3 人与计算机的函数

一定要牢记一点，函数不只是面向计算机的，同时也是面向人的。R 并不在意函数的名称是什么，或者其中有多少注释，但是，这些对于人类读者来说都是非常重要的。本节将讨论编写函数时的一些注意事项，它们可以让函数更容易为人们所理解。

函数名是非常重要的。理想的函数名应该既简短，又能清楚地说明函数的作用。这很难做到！但清楚比简短更重要，因为 RStudio 中的自动完成功能可以使得长函数名更容易输入。

一般来说，函数名应该是动词，而参数名应该是名词。但也有例外：如果函数计算的是一个众所周知的名词（比如，mean() 要比 compute_mean() 好），或者读取的是对象的某种属性（比如，coef() 要比 get_coefficients() 好）时，那么函数名使用名词也是可以的。如果使用的是非常广义的动词，如 get、compute、calculate 或 determine，此时名词通常是更好的选择。尽量发挥你的判断力，如果发现更好的函数名称，那么就赶快改名吧，不要犹豫：

```
# 名称太短
f()

# 名称不是动词，或者没有描述力
my_awesome_function()

# 名称虽然长，但是表达得很清楚
impute_missing()
collapse_years()
```

如果你的函数名由多个单词组成，那么我们建议使用“snake_case”命名法，即使用小写单词，单词之间用下划线隔开。另一种常用的命名法是“camelCase”，即首个单词小写，其余单词首字母大写。其实选择哪种命名法并不重要，重要的是要保持一致，选择一种方法并坚持使用即可。R 本身就不太一致，但我们对此无能为力。为了避免重蹈覆辙，一定要尽量让自己的代码具有一致性：

```
# 千万别这样!
col_mins <- function(x, y) {}
rowMaxes <- function(y, x) {}
```

如果你有一族功能相似的函数，那么一定要确保它们具有一致的名称和参数。可以使用一个通用前缀来表明它们之间的联系，这种方式比使用通用后缀更好，因为如果 IDE 有自动完成功能，那么就可以在输入前缀后列举出这个函数族中的所有成员：

```
# 良好的命名方式
input_select()
input_checkbox()
input_text()
# 不太好的命名方式
select_input()
checkbox_input()
text_input()
```

stringr 包就是这种设计的一个优秀示例：如果没有确切地记住所需的函数名称，那么可以先输入 str_，然后就能唤起你失落的记忆了。

应该尽可能避免覆盖现有的函数和变量。总体来说，完全不覆盖是不可能的，因为太多好名称已经被其他 R 包占用了，但完全可以不覆盖 R 基础包中最常用的名称，这样可以避免混淆：

```
# 不要这样!
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

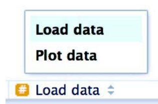
你应该使用注释（即由 # 开头的行）来解释代码，需要解释的是“为什么”，一般不用解释“是什么”或“如何做”。如果通过阅读无法理解代码的行为，那么就应该考虑如何重写代码才能让它更清晰。是否应该添加一些名称有意义的中间变量？是否应该从一个较大的函数中分解出一部分代码，将其转换为一个有意义的函数？但是，代码永远不能表示出决策背后的思考过程：为什么我们选择这种方法，而不是另一种？如果这种方法不奏效，那我们还能怎么做？在注释中表达这种思考过程是一种非常好的做法。

注释的另一个重要作用是对代码进行分节，从而使其更易读。可以使用一长串 - 或 = 让代码各节之间的界限更加明显：

```
# 加载数据 --------------------------------------

# 绘制数据 --------------------------------------
```

RStudio 提供了键盘快捷方式来创建这种分节标记，即 Ctrl+Shift+R，并且可以在脚本编辑器左下角的代码浏览弹出菜单中显示这些标记：



练习

(1) 阅读以下 3 个函数的源代码，推测出它们的功能，然后使用头脑风暴给它们以更好的名称。

```
f1 <- function(string, prefix) {
  substr(string, 1, nchar(prefix)) == prefix
}
f2 <- function(x) {
  if (length(x) <= 1) return(NULL)
  x[-length(x)]
}
f3 <- function(x, y) {
  rep(y, length.out = length(x))
}
```

(2) 找一个你最近编写的函数，然后花 5 分钟进行头脑风暴，给它及其参数一个更好的名称。

(3) 比较并对比 rnorm() 和 MASS::mvrnorm() 函数，如何才能使得它们更加一致？

(4) 举例说明 norm_r()、norm_d() 等函数比 rnorm() 和 dnorm() 更具优势。再举一个反例说明它们的缺点。

14.4 条件执行

if 语句可以使得你有条件地执行代码。其形式如下所示：

```
if (condition) {
  # 条件为真时执行的代码
```

```
} else {
  # 条件为假时执行的代码
}
```

要想获得有关 if 语句的帮助，你需要使用反引号将其括起来：?`if`。如果不是熟练的程序员，那么这个帮助对你也没什么大用，但你至少应该知道如何获取这个帮助！

下面是使用了 if 语句的一个简单函数。这个函数的目的是返回一个逻辑向量，用于描述一个向量的各个元素是否被命名：

```
has_name <- function(x) {
  nms <- names(x)
  if (is.null(nms)) {
    rep(FALSE, length(x))
  } else {
    !is.na(nms) & nms != ""
  }
}
```

这个函数利用了标准返回值原则，即函数返回其计算的最后一个值。这里是 if 语句的两个分支中的任意一个。

14.4.1 条件

condition 的值要么是 TRUE，要么是 FALSE。如果它是一个向量，那么你会收到一条警告；如果它是 NA，那么程序就会出错。注意你自己代码中的这些消息：

```
if (c(TRUE, FALSE)) {}
#> Warning in if (c(TRUE, FALSE)) {:
#> the condition has length > 1 and only the
#> first element will be used
#> NULL

if (NA) {}
#> Error in if (NA) {: missing value where TRUE/FALSE needed
```

你可以使用 || （或）和 && （与）操作符来组合多个逻辑表达式。这些操作符具有“短路效应”：只要 || 遇到第一个 TRUE，那么就会返回 TRUE，不再计算其他表达式；只要 && 遇到第一个 FALSE，就会返回 FALSE，不再计算其他表达式。不能在 if 语句中使用 | 或 &，它们是向量化的操作符，只可以用于多个值（这就是我们在 filter() 函数中使用它们的原因）。如果一定要使用逻辑向量，那么你可以使用 any() 或 all() 函数将其转换为单个逻辑值。

在测试相等关系时，一定要小心，== 是向量化的，很容易输出多个值。你要么先检查结果的长度是否为 1，然后使用 all() 或 any() 函数进行转换；要么使用非向量化的 identical() 函数。identical() 非常严格，总是返回一个 TRUE 或者一个 FALSE，并且不限制参数类型。这意味着，在比较整数和双精度数时，一定要注意：

```
identical(0L, 0)
#> [1] FALSE
```

你还需要提防浮点数的问题：

```
x <- sqrt(2) ^ 2
x
#> [1] 2
x == 2
#> [1] FALSE
x - 2
#> [1] 4.44e-16
```

解决方式是使用 `dplyr::near()` 函数进行比较，我们在 3.2.1 节中介绍过这个函数。

记住，`x == NA` 没有任何作用。

14.4.2 多重条件

你可以将多个 `if` 语句串联起来：

```
if (this) {
  # 做一些操作
} else if (that) {
  # 做另外一些操作
} else {
  #
}
```

但如果你有一长串 `if` 语句，那么就要考虑重写了。重写的一种方法是使用 `switch()` 函数，它先对第一个参数求值，然后按照名称或位置在后面的参数列表中匹配返回结果：

```
#> function(x, y, op) {
#>   switch(op,
#>     plus = x + y,
#>     minus = x - y,
#>     times = x * y,
#>     divide = x / y,
#>     stop("Unknown op!")
#>   )
#> }
```

可以重写一长串 `if` 语句的另一个函数是 `cut()`，它可以将连续变量离散化。

14.4.3 代码风格

`if` 和 `function` 后面总是要跟着一对大括号（`{}`），其中的内容应该缩进两个空格。这样通过左侧空白就可以很容易地知道代码层次。

左大括号不应该自己占一行，而且后面要换行。右大括号应该自己占一行，除非后面跟着 `else`。大括号中的代码一定要缩进：

```
# 好
if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
```

```
  log(x)
} else {
  y ^ x
}

# 不好
if (y < 0 && debug)
message("Y is negative")

if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

如果 `if` 语句非常短，可以在一行内写下，那么可以不用大括号：

```
y <- 10
x <- if (y < 20) "Too low" else "Too high"
```

我们建议只对特别短的 `if` 语句采用这种形式，其他情况下还是完整形式更易于阅读：

```
if (y < 20) {
  x <- "Too low"
} else {
  x <- "Too high"
}
```

14.4.4 练习

(1) `if` 与 `ifelse()` 的区别是什么？仔细阅读帮助文档，然后构建 3 个示例，说明它们之间的关键区别。

(2) 编写一个欢迎函数，根据每天的不同时间输出“上午好”“下午好”和“晚上好”。（提示：使用 `lubridate::now()` 函数默认的时间参数，这会使得函数测试更容易一些。）

(3) 实现 `fizzbuzz` 函数，接受一个数值作为输入。如果这个数值能被 3 整除，那么就返回“fizz”；如果能被 5 整除，就返回“buzz”；如果能同时被 3 和 5 整除，则返回“fizzbuzz”；否则，就返回这个数值。一定要先调试代码，再创建函数。

(4) 如何使用 `cut()` 函数来简化以下这段嵌套 if-else 语句？

```
if (temp <= 0) {
  "freezing"
} else if (temp <= 10) {
  "cold"
} else if (temp <= 20) {
  "cool"
} else if (temp <= 30) {
  "warm"
} else {
  "hot"
}
```

如果以上代码中使用的是 <，不是 <=，那么应该如何修改 cut() 函数的调用方式？对于这个问题，cut() 函数的其他主要优点是什么？（提示：如果 temp 中有多个值，那么会出现什么情况？）

(5) 如果 switch() 函数和数值一起使用，那么会是什么情况？

(6) 以下 switch() 函数的作用是什么？如果 x 是 e，那么会出现什么情况？

```
switch(x,
  a = ,
  b = "ab",
  c = ,
  d = "cd"
)
```

进行实测，然后仔细阅读文档。

14.5 函数参数

函数的参数通常分为两大类：一类提供需要进行计算的**数据**，另一类控制计算过程的**细节**。举例如下。

- 在 log() 函数中，数据是 x，细节则是对数的底，即 base。
- 在 mean() 函数中，数据是 x，细节则是从 x 前后两端（trim）移除多大比例的数据，以及如何处理缺失值（na.rm）。
- 在 t.test() 函数中，数据是 x 和 y，检验的细节则是 alternative、mu、paired、var.equal 以及 conf.level 等设置。
- 在 str_c() 函数中，你可以向 ... 参数提供任意数量的字符串作为数据，连接的细节则由 sep 和 collapse 参数控制。

通常情况下，数据参数应该放在最前面，细节参数则放在后面，而且一般都有默认值。设置默认值的方式与使用命名参数调用函数的方式是一样的：

```
# 使用近似正态分布计算均值两端的置信区间
mean_ci <- function(x, conf = 0.95) {
  se <- sd(x) / sqrt(length(x))
  alpha <- 1 - conf
  mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))
}

x <- runif(100)
mean_ci(x)
#> [1] 0.498 0.610
mean_ci(x, conf = 0.99)
#> [1] 0.480 0.628
```

默认值应该几乎总是最常用的值。这种原则的例外情况非常少，除非出于安全考虑。例如，将 na.rm 的默认值设为 FALSE 是情有可原的，因为缺失值有时是非常重要的。虽然代码中经常使用的是 na.rm = TRUE，但是通过默认设置不声不响地忽略缺失值并不是一种良好的做法。

在调用函数时，我们经常省略数据参数的名称，因为其使用太普遍了。如果不想使用细节参数的默认值，那么你应该使用细节参数的完整名称：

```
# 好
mean(1:10, na.rm = TRUE)

# 不好
mean(x = 1:10, , FALSE)
mean(, TRUE, x = c(1:10, NA))
```

如果一个参数名称的前几个字母可以唯一标识这个参数，那么你可以通过这些字母来引用这个参数，如 `mean(x, n = TRUE)`。但是，通常要在不会引起混淆的情况下才能这样做。

注意，在调用函数时，应该在其中 `=` 的两端都加一个空格。逗号后面应该总是加一个空格，逗号前面则不要加空格（与英文写法相同）。使用空格可以使得函数的重要部分更易读：

```
# 好
average <- mean(feet / 12 + inches, na.rm = TRUE)

# 不好
average<-mean(feet/12+inches,na.rm=TRUE)
```

14.5.1　选择参数名称

参数名称也很重要。R 并不在乎参数名称，但代码的读者（包括未来的你）不能不在乎！通常应该选择那些较长的、更具描述性的名称，但 R 中有一些非常短的通用名称，你应该记住它们。

- `x`，`y`，`z`：向量。
- `w`：权重向量。
- `df`：数据框。
- `i`，`j`：数值索引（通常用于表示行和列）。
- `n`：长度或行的数量。
- `p`：列的数量。

除此之外，你还可以考虑使用现有 R 函数中的参数名称。例如，使用 `na.rm` 来确定是否需要除去缺失值。

14.5.2　检查参数值

当编写的函数越来越多时，你有时会记不清某个函数到底是用来做什么的。这时就很容易使用无效的参数来调用函数。为了解决这种问题，应该对函数参数进行明确的限制。例如，假设你已经编写了一些函数来计算加权摘要统计量：

```
wt_mean <- function(x, w) {
  sum(x * w) / sum(x)
}
wt_var <- function(x, w) {
  mu <- wt_mean(x, w)
  sum(w * (x - mu) ^ 2) / sum(w)
```

```
}
wt_sd <- function(x, w) {
  sqrt(wt_var(x, w))
}
```

如果 x 和 w 的长度不一样，那么会发生什么情况？

```
wt_mean(1:6, 1:3)
#> [1] 2.19
```

这种情况下，由于 R 的向量循环机制，代码不会出错。

对重要的前提条件进行检查，当其不为真时就抛出一个错误（使用 stop() 函数），这是一种非常好的做法：

```
wt_mean <- function(x, w) {
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }
  sum(w * x) / sum(x)
}
```

当心，别做得太过头。你应该权衡在编写函数上要花费的时间与让函数更健壮要花费的时间。例如，如果你还需要添加一个 na.rm 参数，那么我们大概不会检查得如此仔细：

```
wt_mean <- function(x, w, na.rm = FALSE) {
  if (!is.logical(na.rm)) {
    stop("`na.rm` must be logical")
  }
  if (length(na.rm) != 1) {
    stop("`na.rm` must be length 1")
  }
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }

  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
  }
  sum(w * x) / sum(x)
}
```

这就有点事倍功半，得不偿失了。比较好的一个折中方案是使用内置的 stopifnot() 函数，它会检查每个参数是否为真，如果某个参数不为真，则生成一条通用的错误消息：

```
wt_mean <- function(x, w, na.rm = FALSE) {
  stopifnot(is.logical(na.rm), length(na.rm) == 1)
  stopifnot(length(x) == length(w))

  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
```

```
  }
  sum(w * x) / sum(x)
}
wt_mean(1:6, 6:1, na.rm = "foo")
#> Error: is.logical(na.rm) is not TRUE
```

注意，如果使用了 `stopifnot()` 函数，那么你实际上是断言了哪些参数必须为真，而不是检查哪些参数可能是错的。

14.5.3　点点点（...）

R 中的很多函数可以接受任意数量的输入：

```
sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
#> [1] 55
stringr::str_c("a", "b", "c", "d", "e", "f")
#> [1] "abcdef"
```

这种函数是如何运行的呢？它们需要一个特殊参数：`...`（读作点点点）。这个特殊参数会捕获任意数量的未匹配参数。

这个参数的作用非常大，因为你可以将它捕获的值传给另一个函数。如果你的函数是另一个函数的包装器，那么这种一网打尽的方式就非常有用了。例如，我们经常用以下方式创建辅助函数来包装 `str_c()` 函数：

```
commas <- function(...) stringr::str_c(..., collapse = ", ")
commas(letters[1:10])
#> [1] "a, b, c, d, e, f, g, h, i, j"

rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <- getOption("width") - nchar(title) - 5
  cat(title, " ", stringr::str_dup(pad, width), "\n", sep = "")
}
rule("Important output")
#> Important output ---------------------------------------
```

这里 `...` 可以将我们不想处理的所有参数传递给 `str_c()`。虽然非常方便，但这种技术是有代价的：所有拼写错误的参数都不会引发错误消息。这使得我们很难发现输入错误：

```
x <- c(1, 2)
sum(x, na.mr = TRUE)
#> [1] 4
```

如果想要检查 `...` 中的值，那么你可以使用 `list(...)`。

14.5.4　惰性求值

R 中的参数求值的方式是惰性的，即直到需要参数时才会进行求值。这意味着，如果没有使用参数，那么它就一直没有实际值。这是 R 作为编程语言的一个重要特性，但在编写函数进行数据分析时，这个特性一般是不重要的。如果想要了解惰性求值的更多知识，参考 http://adv-r.had.co.nz/Functions.html#lazy-evaluation。

14.5.5 练习

(1) 函数调用 `commas(letters, collapse = "-")` 的作用是什么？为什么？

(2) 如果能为 `pad` 参数提供多个字符，那真是太好了，例如 `rule("Title", pad = "-+")`。为什么现在的 `rule()` 函数还做不到这一点？应该如何改进？

(3) `mean()` 函数中的 `trim` 参数的作用是什么？何时应该使用这个参数？

(4) `cor()` 函数的参数 `method` 的默认值是 `c("pearson", "kendall", "spearman")`，这个默认值的意义是什么？默认情况下会使用哪个值？

14.6 返回值

弄清楚函数要返回什么结果通常是非常简单的一件事：其实就是当初创建这个函数的原因！对于返回值，以下两个问题需要你仔细思考。

- 提前返回能否让函数更易读？
- 你能使得自己的函数支持管道操作吗？

14.6.1 显式返回语句

函数的返回值通常是最后一个语句的值，但你可以通过 `return()` 语句提前返回一个值。我们认为最好有节制地使用 `return()` 语句，因为提前返回的一般都是比较简单的情况。常见的提前返回原因就是输入为空：

```
complicated_function <- function(x, y, z) {
  if (length(x) == 0 || length(y) == 0) {
    return(0)
  }

  # 这里是复杂的代码
}
```

需要提前返回的另一个原因是，`if` 语句的一个分支非常复杂，而另一个分支则特别简单。例如，你可能写出如下的 `if` 语句：

```
f <- function() {
  if (x) {
    # 需要
    # 多行
    # 代码
    # 才能
    # 完成
    # 的
    # 操作
    # express
  } else {
    # 返回一个非常简单的值
  }
}
```

但如果第一个分支中的代码非常长，到达 else 语句前，你可能就已经记不清 condition 了。解决这个问题的一种方法是将简单情形提前返回：

```
f <- function() {
  if (!x) {
    return(something_short)
  }

  # 需要
  # 多行
  # 代码
  # 才能
  # 完成
  # 的
  # 操作
  # express
}
```

这样应该会使得代码更容易理解，因为不需要太多的上下文。

14.6.2 使得函数支持管道

如果想要让自己的函数支持管道操作，那么你应该仔细思考一下返回值。可以支持管道操作的函数有两种主要类型：转换函数与副作用函数。

转换函数会传入一个明确的“基本”对象作为第一个参数，对这个对象进行处理后，再将其返回。例如，在 dplyr 中，这个关键的对象就是数据框。如果能够确定在自己的领域内应该使用哪种数据类型，那么你就可以让自己的函数支持管道操作了。

副作用函数经常用来执行某种行为，比如绘图或保存文件，而不是转换对象。这些函数会“悄悄地”返回第一个参数，因此，默认情况下，第一个参数不显示在输出中，但仍然可以由管道操作使用。例如，以下这个简单函数会输出一个数据框中的缺失值的数量：

```
show_missings <- function(df) {
  n <- sum(is.na(df))
  cat("Missing values: ", n, "\n", sep = "")

  invisible(df)
}
```

如果以交互式方式调用，那么 invisible() 函数的作用就是使得输入参数 df 不显示在输出中：

```
show_missings(mtcars)
#> Missing values: 0
```

但是 df 确实是在输出中的，只是默认不显示：

```
x <- show_missings(mtcars)
#> Missing values: 0
class(x)
#> [1] "data.frame"
dim(x)
#> [1] 32 11
```

而且我们还可以在管道中使用它：

```
mtcars %>%
  show_missings() %>%
  mutate(mpg = ifelse(mpg < 20, NA, mpg)) %>%
  show_missings()
#> Missing values: 0
#> Missing values: 18
```

14.7 环境

本章的最后内容是函数的环境。当刚开始编写函数时，不需要对环境有多深入的理解。但我们还是应该了解一些关于环境的知识，因为这些知识对于理解函数如何运行非常重要。函数的环境决定了 R 如何寻找对象的值。例如，查看以下函数：

```
f <- function(x) {
  x + y
}
```

在很多编程语言中，这段代码会引发一个错误，因为函数没有定义 y。这种代码在 R 中是有效的，因为 R 使用称为**词法定界**的一种规则来搜索对象的值。因为 y 没有在函数中进行定义，所以 R 会在定义函数的**环境**中寻找 y：

```
y <- 100
f(10)
#> [1] 110

y <- 1000
f(10)
#> [1] 1010
```

这种方式似乎是解决程序 bug 的一个秘诀，但实际上最好不要故意写出这种函数。总体来说，这种代码不会引起太多问题（特别是为了得到干净的环境而经常重新启动 R 时）。

从编程语言的角度来看，这种方式的优点是可以使得 R 非常一致，因为使用同样的规则来搜索所有对象。对于函数 f()，你可以通过函数定义让 { 和 + 具有一些出人意料的行为，比如以下这个恶作剧：

```
`+` <- function(x, y) {
  if (runif(1) < 0.1) {
    sum(x, y)
  } else {
    sum(x, y) * 1.1
  }
}
table(replicate(1000, 1 + 2))
#>
#>   3 3.3
#> 100 900
rm(`+`)
```

这种重定义在 R 中是很普遍的一种现象。R 很少对编程能力进行限制，你可以做很多其他语言中无法进行的事情，甚至是 99% 的人都认为极不明智的那些事情（比如重定义加法运算）。但是，正是这种能力和灵活性才使得 ggplot2 和 dplyr 工具开发成为可能。如何充分利用这种灵活性已经超出了本书范围，如果想要学习这方面的知识，可以参考 *Advanced R*。

第 15 章

向量

15.1 简介

到目前为止，我们已经介绍过了 tibble 以及支持 tibble 的 R 包。但是，当开始编写函数或者更加深入地学习 R 时，你就会发现需要使用构成 tibble 的基础对象，即向量。如果曾经以更加传统的方式学习过 R，那么你应该已经对向量非常熟悉，因为多数 R 学习资源都是先介绍向量，然后才扩展到 tibble 的。我们则认为应该从 tibble 开始，因为它是即学即用的，熟悉 tibble 后，再来学习构成它的基础成分比较好。

向量特别重要，因为绝大多数自定义函数都要使用向量。也可以直接编写使用 tibble 的函数（比如 ggplot2 和 dplyr 中的函数），但从目前来看，支持这种函数编写方式的工具还很不成熟。我们正在开发一种更好的工具，参见 https://github.com/hadley/lazyeval。即使有了更好的工具，你还是需要理解向量，因为使用向量更容易开发出用户友好的高阶函数。

准备工作

本章的重点在于介绍 R 基础包中的数据结构，因此无须加载任何 R 包。但是，为了避免 R 基础包中的一些不一致现象，我们会使用 purrr 包中的少量函数。

```
library(tidyverse)
#> Loading tidyverse: ggplot2
#> Loading tidyverse: tibble
#> Loading tidyverse: readr
#> Loading tidyverse: purrr
#> Loading tidyverse: dplyr
#> Conflicts with tidy packages --------------------------------
#> filter(): dplyr, stats
#> lag():    dplyr, stats
```

15.2　向量基础

向量的类型主要有两种。

- **原子向量**，其共有 6 种类型：**逻辑型**、**整型**、**双精度型**、**字符型**、**复数型**和**原始型**。整型和双精度型向量又统称为**数值型**向量。
- **列表**，有时又称为递归向量，因为列表中也可以包含其他列表。

原子向量与列表之间的主要区别是，原子向量中的各个值都是**同种类型的**，而列表中的各个值可以是**不同类型的**。`NULL` 是一个与向量相关的对象，用于表示空向量（与表示向量中的一个值为空的 `NA` 不同），通常指长度为 0 的向量。图 15-1 总结了各种向量之间的联系。

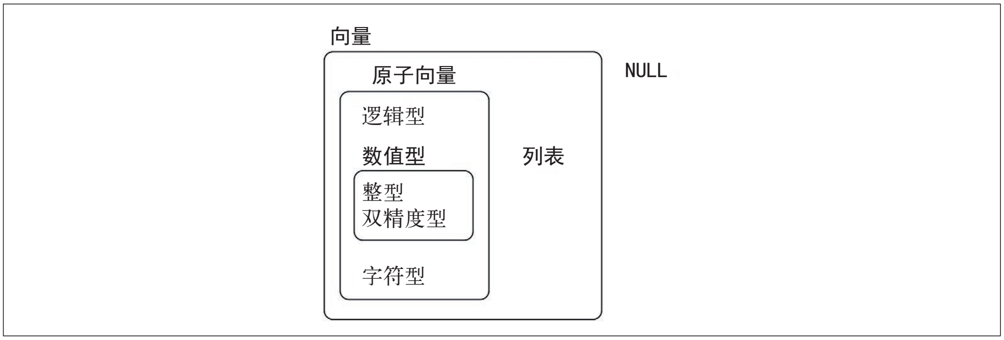


图 15-1：R 向量类型的层次图

每个向量都有两个关键属性。

- **类型**。你可以使用 `typeof()` 函数来确定向量的类型：

```
typeof(letters)
#> [1] "character"
typeof(1:10)
#> [1] "integer"
```

- **长度**。你可以使用 `length()` 函数来确定向量的长度：

```
x <- list("a", "b", 1:10)
length(x)
#> [1] 3
```

还可以向向量中任意添加额外的元数据，这些元数据称为特性。特性可以用来创建**扩展向量**，以执行一些新的操作。比较重要的扩展向量有 4 种类型。

- 基于整型向量构建的因子。
- 基于数值型向量构建的日期和日期时间。
- 基于列表构建的数据框和 tibble。

本章将由浅入深地介绍这些重要向量。首先是原子向量、然后是列表，最后是扩展向量。

15.3 重要的原子向量

4 种最重要的原子向量类型是逻辑型、整型、双精度型和字符型。原始型与复数型很少在数据分析中使用，因此不做介绍。

15.3.1 逻辑型

逻辑型向量是最简单的一种原子向量，因为它们只有 3 个可能的取值：`FALSE`、`TRUE` 和 `NA`。一般可以通过比较运算符来构建逻辑向量，我们已经在 3.2.1 节中介绍过这种运算符。你还可以通过 `c()` 函数来手工创建逻辑向量：

```
1:10 %% 3 == 0
#>  [1] FALSE FALSE  TRUE FALSE FALSE
#>  [2] TRUE FALSE FALSE  TRUE FALSE
c(TRUE, TRUE, FALSE, NA)
#> [1]  TRUE  TRUE FALSE    NA
```

15.3.2 数值型

整型与双精度型向量统称为数值型向量。R 中默认数值是双精度型的。如果想要创建整型数值，可以在数字后面加一个 `L`：

```
typeof(1)
#> [1] "double"
typeof(1L)
#> [1] "integer"
1.5L
#> [1] 1.5
```

整型和双精度型之间的区别一般并不重要，但你需要注意其中两个重要区别。

- 双精度型是近似值。双精度型表示的是浮点数，不能由固定数量的内存精确表示。这意味着你应该将所有双精度数当成近似值。例如，2 的平方根的平方是多少？

  ```
  x <- sqrt(2) ^ 2
  x
  #> [1] 2
  x - 2
  #> [1] 4.44e-16
  ```

  在处理浮点数时，这种现象非常普遍：多数计算都包括一些近似误差。在比较浮点数时，不能使用 `==`，而应该使用 `dplyr::near()`，后者可以容忍一些数据误差。

- 整型数据有 1 个特殊值 `NA`，而双精度型数据则有 4 个特殊值：`NA`、`NaN`、`Inf` 和 `-Inf`。其他 3 个特殊值都可以由除法产生：

  ```
  c(-1, 0, 1) / 0
  #> [1] -Inf  NaN  Inf
  ```

  不要使用 `==` 来检查这些特殊值，而应该使用辅助函数 `is.finite()`、`is.infinite()` 和 `is.nan()`。

	0	Inf	NA	NaN
is.finite()	x			
is.infinite()		x		
is.na()			x	x
is.nan()				x

15.3.3　字符型

字符向量是最复杂的一种原子向量，因为其每个元素都是一个字符串，而字符串可以包含任意数量的数据。

第 10 章中已经介绍了很多处理字符串的方法。这里我们要强调关于字符串基础实现的一个重要特征：R 使用的是全局字符串池。这意味着每个唯一的字符串在内存中只保存一次，每次对这个字符串的使用都指向这段内存，这样可以减少复制字符串所需的内存空间。你可以使用 `pryr::object_size()` 函数来查看这种处理的效果：

```
x <- "This is a reasonably long string."
pryr::object_size(x)
#> 136 B

y <- rep(x, 1000)
pryr::object_size(y)
#> 8.13 kB
```

y 所占用的内存不是 x 的 1000 倍，因为 y 中的每个元素都只是指向一个字符串的指针。因为一个指针占 8 个字节，所以 1000 个指针以及一个 136 字节的字符串所占用的内存空间是 8 * 1000 + 136 = 8.13KB。

15.3.4　缺失值

注意，每种类型的原子向量都有自己的缺失值：

```
NA            # 逻辑型
#> [1] NA
NA_integer_   # 整型
#> [1] NA
NA_real_      # 双精度型
#> [1] NA
NA_character_ # 字符型
#> [1] NA
```

一般不需要考虑类型问题，因为你可以一直使用 `NA`，R 会通过隐含的强制类型转换规则（后面会介绍）将其转换为正确的类型。但是，因为有些函数对输入的要求非常严格，所以你应该了解这些知识，做到有备无患，以便在需要时可以指定相应的类型。

15.3.5　练习

(1) 描述 `is.finite(x)` 和 `!is.infinite(x)` 之间的区别。

(2) 阅读 `dplyr::near()` 函数的源代码（提示：要想看到源代码，去掉 `()`，只输入函数名称即可）。它是如何运行的？

(3) 逻辑向量可以取 3 个可能的值。整数向量有多少个可能的值？双精度向量可以取多少个值？使用 Google 做一些这方面的研究。

(4) 通过头脑风暴，找出至少 4 种将双精度型转换为整型的方法。精确描述它们之间的区别。

(5) readr 包中的哪些函数可以将一个字符串转换为逻辑型、整型和双精度型向量？

15.4　使用原子向量

我们已经弄清楚了不同类型的原子向量间的差别，接下来将讨论处理原子向量的几种重要操作，具体如下。

- 如何将一种原子向量转换为另一种，以及何时系统会自动转换。
- 如何分辨出一个对象是哪种特定类型的向量。
- 在处理长度不同的向量时，会发生什么情况。
- 如何命名向量中的元素。
- 如何提取出感兴趣的元素。

15.4.1　强制转换

将一种类型的向量强制转换成另一种类型的方式有两种。

- 显式强制转换：当调用 `as.logical()`、`as.integer()`、`as.double()` 或 `as.character()` 这样的函数进行转换时，使用的就是显式强制转换。只要发现需要进行显式强制转换，那么你就需要检查一下，看看能否在之前的某个步骤中解决这个问题。例如，你可能需要修改 readr 中的 `col_type` 的设置。
- 隐式强制转换：当在特殊的上下文环境中使用向量，而这个环境又要求使用特定类型的向量时，就会发生隐式强制转换。例如，在数值型摘要函数中使用逻辑向量，或者在需要整型向量的函数中使用了双精度型向量时。

因为显式强制转换使用得相对较少，而且非常容易理解，所以我们重点介绍隐式强制转换。

你已经见过了最重要的一种隐式强制转换：在数值环境中使用逻辑向量。这种情况下，`TRUE` 转换为 1，`FALSE` 转换为 0。这意味着对逻辑向量求和的结果就是其中真值的个数，逻辑向量的均值就是其中真值的比例：

```
x <- sample(20, 100, replace = TRUE)
y <- x > 10
sum(y) # 大于10的数有多少个？
#> [1] 44
mean(y) # 大于10的数的比例是多少？
#> [1] 0.44
```

你或许还会看到（通常是较久远的）有些代码使用了反向的隐式强制转换，即从整型转换

为逻辑型：

```
if (length(x)) {
  # 进行某种操作
}
```

在以上示例中，0 转换为 FALSE，其他任何值都转换为 TRUE。我们认为这样做会让代码更难理解，因此不推荐这种方式，而是应该明确地设定条件：length(x) > 0。

当试图使用 c() 函数来创建包含多种类型元素的向量时，清楚如何进行类型转换也是非常重要的。这时总是会统一转换为最复杂的元素类型：

```
typeof(c(TRUE, 1L))
#> [1] "integer"
typeof(c(1L, 1.5))
#> [1] "double"
typeof(c(1.5, "a"))
#> [1] "character"
```

原子向量中不能包含不同类型的元素，因为类型是整个向量的一种属性，不是其中单个元素的属性。如果需要在同一个向量中包含混合类型的元素，那么就需要使用列表，我们很快就会对其进行介绍。

15.4.2　检验函数

有时我们需要根据向量的类型进行不同的操作。检验向量类型的一种方法是使用 typeof() 函数，另一种方法是使用检验函数来返回 TRUE 或 FALSE。R 基础包中提供了很多这样的函数，如 is.vector() 和 is.atomic()，但它们经常返回出人意料的结果。更可靠的方法是使用 purrr 包提供的 is_* 函数族，以下表格总结了它们的使用方式。

	逻辑型	整型	双精度型	字符型	列表
is_logical()	x				
is_integer()		x			
is_double()			x		
is_numeric()		x	x		
is_character()				x	
is_atomic()	x	x	x	x	
is_list()					x
is_vector()	x	x	x	x	x

以上每个函数都配有一个“标量”版本，比如 is_scalar_atomic()，它们可以检验长度为 1 的向量。这些函数非常有用，例如，如果想要检验函数的一个参数是否为单个逻辑值，那么就可以使用这种函数。

15.4.3　标量与循环规则

R 可以隐式地对向量类型进行强制转换，同样地，也可以对向量长度进行强制转换。这种转换称为**向量循环**，因为 R 会将较短的向量重复（或称循环）到与较长的向量相同的长度。

混合使用向量和“标量”时，向量循环是最有用的。我们在标量上加了引号，因为 R 中没有真正的标量，只有长度为 1 的向量。正因为没有标量，所以 R 的多数内置函数都是**向量化的**，即可以在数值的一个向量上进行操作。这就是以下代码可以运行的原因。

```
sample(10) + 100
#>  [1] 109 108 104 102 103 110 106 107 105 101
runif(10) > 0.5
#>  [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
#>  [10]  TRUE
```

在 R 中，基本的数学运算是使用向量来进行的，这意味着在进行简单数据计算时，根本不需要执行显式迭代。

如果两个长度相同的向量相加，或者一个向量和一个“标量”相加，那么结果是显而易见的，但是，如果两个长度不同的向量相加，那么会出现什么情况呢？

```
1:10 + 1:2
#>  [1]  2  4  4  6  6  8  8 10 10 12
```

这里 R 会扩展较短的向量，使其与较长的向量一样长，这个过程就称作向量循环。这个过程是默默进行的，除非较长向量的长度不是较短向量长度的整数倍：

```
1:10 + 1:3
#> Warning in 1:10 + 1:3:
#> longer object length is not a multiple of shorter
#> object length
#>  [1]  2  4  6  5  7  9  8 10 12 11
```

虽然可以创建非常简洁优雅的代码，但向量循环也可以悄无声息地掩盖某些问题。为此，只要循环的不是一个标量，那么 tidyverse 中的向量化函数就会抛出一条错误消息。如果确实想要执行向量循环，那么你需要使用 `rep()` 函数手工完成：

```
tibble(x = 1:4, y = 1:2)
#> Error: Variables must be length 1 or 4.
#> Problem variables: 'y'

tibble(x = 1:4, y = rep(1:2, 2))
#> # A tibble: 4 × 2
#>       x     y
#>   <int> <int>
#> 1     1     1
#> 2     2     2
#> 3     3     1
#> 4     4     2

tibble(x = 1:4, y = rep(1:2, each = 2))
#> # A tibble: 4 × 2
#>       x     y
#>   <int> <int>
#> 1     1     1
#> 2     2     1
#> 3     3     2
#> 4     4     2
```

15.4.4 向量命名

所有类型的向量都是可以命名的。你可以在使用 c() 函数创建向量时进行命名：

```
c(x = 1, y = 2, z = 4)
#> x y z
#> 1 2 4
```

也可以在向量创建完成后，使用 purrr::set_names() 函数来命名：

```
set_names(1:3, c("a", "b", "c"))
#> a b c
#> 1 2 3
```

命名向量对于接下来要介绍的向量取子集操作特别重要。

15.4.5 向量取子集

我们可以使用 dplyr::filter() 函数在 tibble 中筛选行。但 filter() 函数只能筛选 tibble，因此要想筛选向量，我们需要使用一个新工具：[。[就是取子集函数，调用形式是 x[a]。你可以使用以下 4 种形式来完成向量取子集操作。

- 使用仅包含整数的数值向量。整数要么全部为正数，要么全部为负数，或者为 0。

  使用正整数取子集时，可以保持相应位置的元素：

  ```
  x <- c("one", "two", "three", "four", "five")
  x[c(3, 2, 5)]
  #> [1] "three" "two"   "five"
  ```

  位置可以重复，这样可以生成比输入更长的输出结果：

  ```
  x[c(1, 1, 5, 5, 5, 2)]
  #> [1] "one"  "one"  "five" "five" "five" "two"
  ```

  使用负整数取子集时，会丢弃相应位置的元素：

  ```
  x[c(-1, -3, -5)]
  #> [1] "two"  "four"
  ```

  正数与负数混合使用则会引发一个错误：

  ```
  x[c(1, -1)]
  #> Error in x[c(1, -1)]:
  #> only 0's may be mixed with negative subscripts
  ```

  这条错误消息中提到了使用 0 来取子集，这样不会返回任何值：

  ```
  x[0]
  #> character(0)
  ```

  这种操作一般没什么用处，如果想要创建一个特殊的数据结构来检验函数，那么它或许还有点帮助。

- 使用逻辑向量取子集。这种方式可以提取出 TRUE 值对应的所有元素，一般与比较函数结合起来使用效果最佳：

```
x <- c(10, 3, NA, 5, 8, 1, NA)

# x中的所有非缺失值
x[!is.na(x)]
#> [1] 10  3  5  8  1

# x中的所有偶数值（或缺失值）
x[x %% 2 == 0]
#> [1] 10 NA  8 NA
```

- 如果是命名向量，那么可以使用字符向量来取子集：

```
x <- c(abc = 1, def = 2, xyz = 5)
x[c("xyz", "def")]
#> xyz def
#>   5   2
```

  与使用正整数取子集一样，你也可以使用字符向量重复取出单个元素。

- 取子集的最简方式就是什么都不写：x[]，这样就会返回 x 中的全部元素。这种方式对于向量取子集没有什么用处，但对于矩阵（或其他高维数据结构）取子集则非常重要，因为这样可以取出所有的行或所有的列，只要将行或列保持为空即可。例如，如果 x 是二维的，那么 x[1,] 可以选取出第 1 行和所有列，x[, -1] 则可以选取出所有行和除第 1 列外的所有列。

如果想要学习取子集操作的更多应用，可以参考 *Advanced R* 中的“Subsetting”一章。

[有一个重要的变体 [[。[[从来都是只提取单个元素，并丢弃名称。当想明确表明需要提取单个元素时，你就应该使用 [[，比如在一个 for 循环中。[和 [[的区别对于列表来说更加重要，我们很快就会对其进行讨论。

15.4.6　练习

(1) mean(is.na(x)) 可以告诉你关于向量 x 的何种信息？ sum(!is.finite(x)) 呢？

(2) 仔细阅读 is.vector() 函数的文档，它到底是用于检验什么的？为什么 is.atomic() 函数的结果与前面对原子向量的定义不符？

(3) 比较并对比 setNames() 函数和 purrr::set_names() 函数。

(4) 创建能够接受一个向量作为输入的函数，并返回以下值。

  a. 最后一个元素。应该使用 [还是 [[?
  b. 偶数位置上的元素。
  c. 除最后一个元素外的所有元素。
  d. 仅返回偶数（不包括缺失值）。

(5) 为什么 `x[-which(x > 0)]` 和 `x[x <=0 ]` 不一样？

(6) 如果使用比向量长度大的整数来取子集，那么会发生什么情况？如果使用不存在的名称来取子集，那么又会发生什么情况？

15.5 递归向量（列表）

列表是建立在原子向量基础上的一种复杂形式，因为列表中可以包含其他列表。这种性质使得列表特别适合表示层次结构或树形结构。你可以使用 `list()` 函数创建列表：

```
x <- list(1, 2, 3)
x
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

在处理列表时，`str()` 函数是一个非常有用的工具，因为其重点关注**列表结构**，而不是列表内容：

```
str(x)
#> List of 3
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3

x_named <- list(a = 1, b = 2, c = 3)
str(x_named)
#> List of 3
#>  $ a: num 1
#>  $ b: num 2
#>  $ c: num 3
```

与原子向量不同，`list()` 中可以包含不同类型的对象：

```
y <- list("a", 1L, 1.5, TRUE)
str(y)
#> List of 4
#>  $ : chr "a"
#>  $ : int 1
#>  $ : num 1.5
#>  $ : logi TRUE
```

列表甚至可以包含其他列表！

```
z <- list(list(1, 2), list(3, 4))
str(z)
#> List of 2
#>  $ :List of 2
```

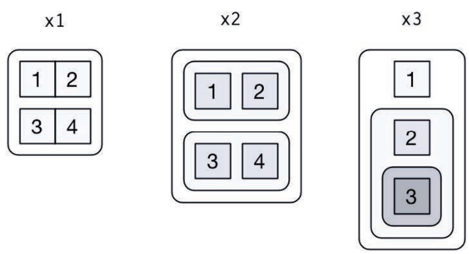
```
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ :List of 2
#>   ..$ : num 3
#>   ..$ : num 4
```

15.5.1 列表可视化

为了解释清楚更复杂的列表操作函数，列表的可视化表示大有裨益。例如，思考以下 3 种列表：

```
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```

我们可以用以下图形来表示它们：



这种可视化表示遵循以下 3 个原则。

- 列表用圆角矩形表示，原子向量用直角矩形表示。
- 子向量绘制在父向量中，而且背景要比父向量深一些，这样更容易表示出层次结构。
- 子向量的方向（也就是其行和列）并不重要，我们只在示例中表示出一行或一列，有时是为了节省空间，有时是为了说明一个重要的属性。

15.5.2 列表取子集

列表取子集有 3 种方式，接下来我们通过列表 a 来说明：

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
```

- 使用 [提取子列表。这种方式的结果总是一个列表：

  ```
  str(a[1:2])
  #> List of 2
  #>  $ a: int [1:3] 1 2 3
  #>  $ b: chr "a string"
  str(a[4])
  #> List of 1
  #>  $ d:List of 2
  #>   ..$ : num -1
  #>   ..$ : num -5
  ```

  和向量一样，你可以使用逻辑向量、整数向量或字符向量来提取子列表。

- 使用 [[从列表中提取单个元素。这种方式会从列表中删除一个层次等级：

```
str(a[[1]])
#>  int [1:3] 1 2 3
str(a[[4]])
#> List of 2
#>  $ : num -1
#>  $ : num -5
```

- $ 是提取列表命名元素的简单方式，其作用与 [[相同，只是不需要使用括号：

```
a$a
#> [1] 1 2 3
a[["a"]]
#> [1] 1 2 3
```

对于列表来说，[和 [[之间的区别是非常重要的，因为 [[会使列表降低一个层级，而 [则会返回一个新的、更小的列表。你可以通过图 15-2 中的可视化表示来比较一下以上代码和输出结果。

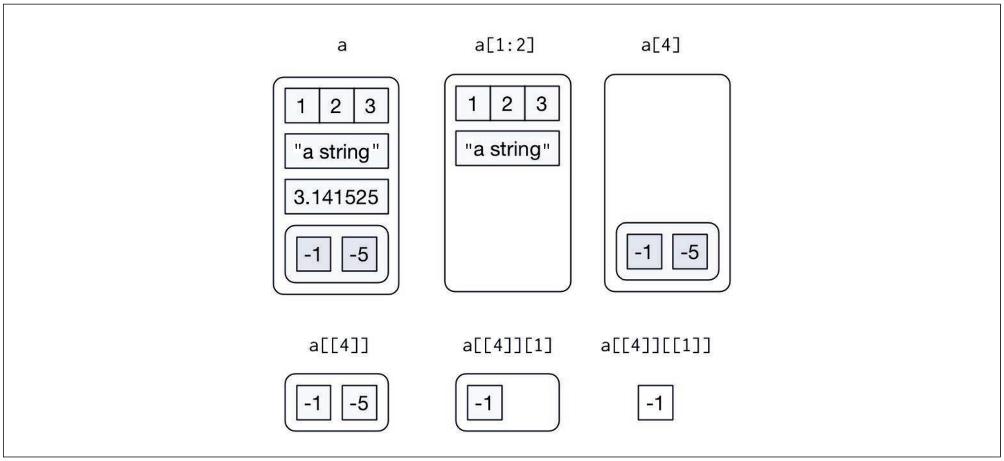


图 15-2：列表取子集的可视化表示

15.5.3 调料列表

[和 [[之间的区别非常重要，但二者很容易混淆。为了帮助记忆，我们使用以下特殊的胡椒罐来模拟列表：



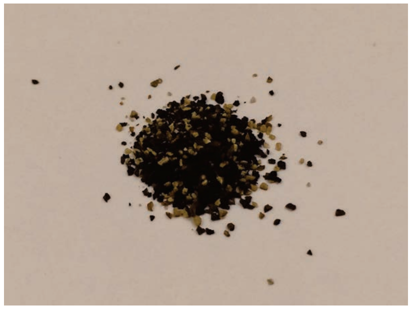
如果这个胡椒罐表示列表 x，那么 x[1] 就是装有一个胡椒袋的胡椒罐：



x[2] 也是一样，但它里面装的是第二个胡椒袋。x[1:2] 就是装有两个胡椒袋的一个胡椒罐。x[[1]] 则是个胡椒袋：



如果想要查看胡椒袋里面的内容，那么你应该使用 x[[1]][[1]]：



15.5.4 练习

(1) 以嵌套集合的方式画出以下列表。

a. list(a, b, list(c, d), list(e, f))
b. list(list(list(list(list(list(a))))))

(2) 如果像对列表取子集一样对 tibble 取子集，那么会发生什么情况？列表和 tibble 之间的关键区别是什么？

15.6 特性

任何向量都可以通过其**特性**来附加任意元数据。你可以将特性看作可以附加在任何对象上的一个向量命名列表。可以使用 attr() 函数来读取和设置单个特性值，也可以使用 attributes() 函数同时查看所有特性值：

```
x <- 1:10
attr(x, "greeting")
#> NULL
attr(x, "greeting") <- "Hi!"
attr(x, "farewell") <- "Bye!"
attributes(x)
#> $greeting
#> [1] "Hi!"
#>
#> $farewell
#> [1] "Bye!"
```

3 种特别重要的特性可以用来实现 R 中的基础功能。

- **名称**：用于命名向量元素。
- **维度**：使得向量可以像矩阵或数组那样操作。
- **类**：用于实现面向对象的 S3 系统。

我们已经在前面介绍过名称了，而且不准备介绍维度，因为本书不会使用矩阵。但我们还是需要描述一下类，因为它可以控制**泛型函数**的运行方式。泛型函数是 R 中实现面向对象

编程的关键，因为它允许函数根据不同类型的输入而进行不同的操作。有关面向对象编程的详细讨论超出了本书范围，但你可以在 *Advanced R* 中找到更多相关内容。

以下就是一个典型的泛型函数：

```
as.Date
#> function (x, ...)
#> UseMethod("as.Date")
#> <bytecode: 0x7fa61e0590d8>
#> <environment: namespace:base>
```

对 `UseMethod` 的调用意味着这是一个泛型函数，而且它会根据第一个参数的类调用特定**方法**，即一个函数。（所有方法都是函数，但函数不一定都是方法。）你可以使用 `methods()` 函数列举出一个泛型函数的所有方法：

```
methods("as.Date")
#> [1] as.Date.character as.Date.date      as.Date.dates
#> [4] as.Date.default as.Date.factor    as.Date.numeric
#> [7] as.Date.POSIXct   as.Date.POSIXlt
#> see '?methods' for accessing help and source code
```

例如，如果 `x` 是一个字符向量，那么 `as.Date()` 就会调用 `as.Date.character()`；如果 `x` 是一个因子，那么就会调用 `as.Date.factor()`。

你可以使用 `getS3method()` 函数查看方法的特殊实现形式：

```
getS3method("as.Date", "default")
#> function (x, ...)
#> {
#>     if (inherits(x, "Date"))
#>         return(x)
#>     if (is.logical(x) && all(is.na(x)))
#>         return(structure(as.numeric(x), class = "Date"))
#>     stop(
#>       gettextf("do not know how to convert '%s' to class %s",
#>       deparse(substitute(x)), dQuote("Date")), domain = NA)
#> }
#> <bytecode: 0x7fa61dd47e78>
#> <environment: namespace:base>
getS3method("as.Date", "numeric")
#> function (x, origin, ...)
#> {
#>     if (missing(origin))
#>         stop("'origin' must be supplied")
#>     as.Date(origin, ...) + x
#> }
#> <bytecode: 0x7fa61dd463b8>
#> <environment: namespace:base>
```

最重要的 S3 泛型函数是 `print()`：当在控制台中输入对象名称时，`print()` 可以决定如何输出这个对象。其他重要的泛型函数是取子集函数 `[`、`[[` 和 `$`。

15.7 扩展向量

原子向量和列表是最基础的向量，使用它们可以构建出另外一些重要的向量类型，比如因子和日期。我们称构建出的这些向量为**扩展向量**，因为它们具有附加**特性**，其中包括类。因为扩展向量中带有类，所以它们的行为就与基础的原子向量不同。本书会使用 4 种重要的扩展向量。

- 因子
- 日期
- 日期时间
- tibble

15.7.1 因子

因子是设计用来表示分类数据的，只能在固定集合中取值。因子是在整型向量的基础上构建的，添加了水平特性：

```
x <- factor(c("ab", "cd", "ab"), levels = c("ab", "cd", "ef"))
typeof(x)
#> [1] "integer"
attributes(x)
#> $levels
#> [1] "ab" "cd" "ef"
#>
#> $class
#> [1] "factor"
```

15.7.2 日期和日期时间

R 中的日期是一种数值型向量，表示从 1970 年 1 月 1 日开始的天数：

```
x <- as.Date("1971-01-01")
unclass(x)
#> [1] 365

typeof(x)
#> [1] "double"
attributes(x)
#> $class
#> [1] "Date"
```

日期时间是带有 `POSIXct` 类的数值型向量，表示从 1970 年 1 月 1 日开始的秒数（POSIXct 表示“可移植操作系统接口”日历时间，portable operating system interface，calendar time。）：

```
x <- lubridate::ymd_hm("1970-01-01 01:00")
unclass(x)
#> [1] 3600
#> attr(,"tzone")
#> [1] "UTC"
typeof(x)
```

```
#> [1] "double"
attributes(x)
#> $tzone
#> [1] "UTC"
#>
#> $class
#> [1] "POSIXct" "POSIXt"
```

tzone 特性是可选的。它控制的是时间的输出方式，不是时间的值：

```
attr(x, "tzone") <- "US/Pacific"
x
#> [1] "1969-12-31 17:00:00 PST"

attr(x, "tzone") <- "US/Eastern"
x
#> [1] "1969-12-31 20:00:00 EST"
```

另一种日期时间类型称为 POSIXlt，它是基于命名列表构建的：

```
y <- as.POSIXlt(x)
typeof(y)
#> [1] "list"
attributes(y)
#> $names
#> [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"
#> [7] "wday"   "yday"   "isdst"  "zone"   "gmtoff"
#>
#> $class
#> [1] "POSIXlt" "POSIXt"
#>
#> $tzone
#> [1] "US/Eastern" "EST"        "EDT"
```

POSIXlt 很少在 tidyverse 中使用，但 R 基础包中确实有这种类型，当需要从日期中提取特定成分（比如年或月）时，就可以使用这种类型。因为 lubridate 已经提供了辅助函数来完成这种操作，所以我们不需要它们。POSIXct 处理起来总是更容易，因此如果遇到了 POSIXlt，那么你就应该使用 lubridate::as_date_time() 将其转换为常用的日期时间。

15.7.3 tibble

tibble 是扩展的列表，有 3 个类：tbl_df、tbl 和 data.frame。它的特性有 2 个：（列）names 和 row.names。

```
tb <- tibble::tibble(x = 1:5, y = 5:1)
typeof(tb)
#> [1] "list"
attributes(tb)
#> $names
#> [1] "x" "y"
#>
#> $class
#> [1] "tbl_df"     "tbl"        "data.frame"
```

```
#>
#> $row.names
#> [1] 1 2 3 4 5
```

传统 data.frames 具有非常相似的结构：

```
df <- data.frame(x = 1:5, y = 5:1)
typeof(df)
#> [1] "list"
attributes(df)
#> $names
#> [1] "x" "y"
#>
#> $row.names
#> [1] 1 2 3 4 5
#>
#> $class
#> [1] "data.frame"
```

以上二者的主要区别就是类。tibble 的类包括了 data.frame，这说明 tibble 自动继承了普通数据框的行为。

tibble（或数据框）与列表之间的主要区别就是 tibble（或数据框）中的所有元素都必须是长度相同的向量。支持 tibble 的所有函数都强制要求这个条件。

15.7.4　练习

(1) hms::hms(3600) 会返回什么？返回值如何输出？这种扩展向量是基于哪种基本类型构造的？使用了哪种特性？

(2) 试着使用长度不同的列创建一个 tibble，会发生什么情况？

(3) 基于前面的定义，列表能否作为 tibble 中的列呢？

第 16 章

使用purrr实现迭代

16.1 简介

第 14 章中讨论了通过创建函数而不是复制粘贴的方式来减少重复代码的重要性。减少重复代码主要有 3 个好处。

- 更容易看清代码的意图，因为吸引我们目光的是那些不同的部分，而不是那些保持不变的部分。
- 更容易对需求变化作出反应。当要修改代码时，只需要在一处进行修改即可，无须对所有复制粘贴的代码都进行修改。
- 更容易减少程序 bug，因为每行代码都被多次使用。

函数是减少重复代码的一种工具，其减少重复代码的方法是，先识别出代码中的重复模式，然后将其提取出来，成为更容易修改和重用的独立部分。减少重复代码的另一种工具是**迭代**，它的作用在于可以对多个输入执行同一种处理，比如对多个列或多个数据集进行同样的操作。本章将介绍两种重要的迭代方式：命令式编程和函数式编程。对于命令式编程，我们将介绍 `for` 循环和 `while` 循环，它们是很好的学习起点，因为这种迭代过程非常简洁明了。但是，`for` 循环比较繁琐，而且每个 `for` 循环中都要重复使用一些记录和跟踪的代码。函数式编程则可以将这些重复代码提取出来，使每个普通的 `for` 循环都可以通过函数来完成。一旦掌握函数式编程的使用方法，那么你就可以通过更少的代码和更容易的方式解决很多常见的迭代问题，出错的概率也会更小。

准备工作

如果已经掌握了 R 基础包中的 `for` 循环，那么你就可以继续学习由 purrr 包提供的一些更加强大的编程工具。purrr 包是 tidyverse 的核心 R 包之一。

```
library(tidyverse)
```

16.2 for循环

假设我们有以下这样一个简单的 tibble：

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

我们想要计算出每列的中位数。你完全可以使用复制粘贴来完成这个任务：

```
median(df$a)
#> [1] -0.246
median(df$b)
#> [1] -0.287
median(df$c)
#> [1] -0.0567
median(df$d)
#> [1] 0.144
```

但这样做就违反了我们的经验法则：永远不要复制粘贴超过 2 次。相反，我们应该使用 `for` 循环：

```
output <- vector("double", ncol(df))  # 1. 输出
for (i in seq_along(df)) {            # 2. 序列
  output[[i]] <- median(df[[i]])      # 3. 循环体
}
output
#> [1] -0.2458 -0.2873 -0.0567 0.1443
```

每个 `for` 循环都包括 3 个部分。

输出：`output <- vector("double", length(x))`

在开始循环前，你必须为输出结果分配足够的空间。这对循环效率非常重要，如果在每次迭代中都使用 `c()` 来保存循环的结果，那么 `for` 循环的速度就会特别慢。

创建给定长度的空向量的一般方法是使用 `vector()` 函数，该函数有两个参数：向量类型（`"logical"`、`"integer"`、`"double"`、`"character"` 等）和向量的长度。

序列：`i in seq_along(df)`

这部分确定了使用哪些值来进行循环：每一轮 `for` 循环都会赋予 `i` 一个来自于 `seq_along(df)` 的不同的值。我们可以将 `i` 看作一个代词，和 it 类似。

你可能还不清楚 `seq_along()` 函数的作用，它与我们熟悉的 `1:length(l)` 的作用基本相同，但最重要的区别是更加安全。如果我们有一个长度为 0 的向量，那么 `seq_along()` 会进行正确的处理：

```
y <- vector("double", 0)
seq_along(y)
#> integer(0)
```

```
1:length(y)
#> [1] 1 0
```

你当然不会故意创建长度为 0 的向量，但很可能会意外生成一个。如果使用 `1:length(x)` 而不是 `seq_along(x)`，那么就可能收到一个令人迷惑的错误消息。

循环体：`output[[i]] <- median(df[[i]])`

这部分就是执行具体操作的代码。它们会重复运行，每次运行都使用一个不同的 `i` 值。第一次迭代运行的是 `output[[1]] <- median(df[[1]])`，第二次迭代运行的是 `output[[2]] <- median[[2]]`，以此类推。

关于 `for` 循环，能讲的就这么多了！现在我们应该使用以下的习题创建一些基础（和不那么基础）的 `for` 循环来练习一下，然后就可以通过 `for` 循环的各种变体来解决实际工作中遇到的那些问题了。

练习

(1) 使用 `for` 循环完成以下操作。

a. 计算出 `mtcars` 数据集中每列的均值。
b. 确定 `nycflights13::flights` 数据集中每列的类型。
c. 计算出 `iris` 数据集中每列唯一值的数量。
d. 分别使用 $\mu = -10$、0、10 和 100 的正态分布生成 10 个随机数。

在编写代码**前**，仔细思考各个 `for` 循环的输出、序列和循环体。

(2) 使用支持向量运算的现有函数替换以下示例中的 `for` 循环。

```
out <- ""
for (x in letters) {
  out <- stringr::str_c(out, x)
}

x <- sample(100)
sd <- 0
for (i in seq_along(x)) {
  sd <- sd + (x[i] - mean(x)) ^ 2
}
sd <- sqrt(sd / (length(x) - 1))

x <- runif(100)
out <- vector("numeric", length(x))
out[1] <- x[1]
for (i in 2:length(x)) {
  out[i] <- out[i - 1] + x[i]
}
```

(3) 使用函数和 `for` 循环完成以下任务。

a. 使用 `for` 循环和 `prints()` 打印出儿歌“Alice the Camel”的歌词。

b. 将儿歌“Ten in the Bed”转换成一个函数，将其扩展为任意数量的小朋友和任意种类的寝具。
c. 将歌曲“99 Bottles of Beer on the Wall”转换成一个函数，将其扩展为任意数量、任意容器、任意液体和任意表面。

(4) 我们经常见到并不预先分配好输出空间，而是在每次循环中增加向量长度的 for 循环，比如：

```
output <- vector("integer", 0)
for (i in seq_along(x)) {
  output <- c(output, lengths(x[[i]]))
}
output
```

这种方式是如何影响性能的？设计一个实验说明一下。

16.3 for循环的变体

如果已经掌握了基础的 for 循环，那么你就应该再熟悉一下它的几种变体。不管进行何种迭代，这些变体都非常重要。因此，即使在下一节中掌握了函数式编程技术，也不要忘了如何使用这些变体。

在基础 for 循环之上有 4 种变体。

- 修改现有对象，而不是创建新对象。
- 使用名称或值进行迭代，而不是使用索引。
- 处理未知长度的输出。
- 处理未知长度的序列。

16.3.1 修改现有对象

有时我们会希望使用 for 循环来修改现有的对象。例如，回想一下第 14 章中的一个问题，我们希望对数据框中的每列进行调整：

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

为了使用 for 循环解决这个问题，我们还是先思考一下 for 循环的 3 个部分。

输出：

我们已经有了输出，和输入是相同的！

序列：

我们可以将数据框看作数据列的列表，因此可以使用 seq_along(df) 在每列中进行迭代。

函数体：

可以使用 rescale01() 函数。

因此可以写出以下代码：

```
for (i in seq_along(df)) {
  df[[i]] <- rescale01(df[[i]])
}
```

一般来说，你可以使用类似的循环来修改列表或数据框，要记住使用 [[，而不是 [。你或许已经发现了，我们在所有 for 循环中使用的都是 [[。我们认为甚至在原子向量中最好也使用 [[，因为它可以明确表示我们要处理的是单个元素。

16.3.2 循环模式

对向量进行循环的基本方式有 3 种，至此我们只介绍了最常用的一种方式：通过 for (i in seq_along(xs)) 使用数值索引进行循环，并使用 x[[i]] 提取出相应的值。另外两种循环方式如下。

- 使用元素进行循环：for (x in xs)。如果只关心副作用，比如绘图或保存文件，那么这种方式是最适合的，因为有效率地保存输出结果是非常困难的。
- 使用名称进行循环：for (nm in names(xs))。这种方式会给出一个名称，你可以使用这个名称和 x[[nm]] 来访问元素的值。如果想要在图表标题或文件名中使用元素名称，那么你就应该使用这种方式。

  如果想要创建命名的输出向量，请一定按照如下方式进行命名：

  ```
  results <- vector("list", length(x))
  names(results) <- names(x)
  ```

使用数值索引进行循环是最常用的方式，因为给定位置后，就可以提取出元素的名称和值：

```
for (i in seq_along(x)) {
  name <- names(x)[[i]]
  value <- x[[i]]
}
```

16.3.3 未知的输出长度

有时你可能不知道输出的长度。例如，假设你想模拟长度随机的一些随机向量。你或许想要通过逐渐增加向量长度的方式来解决这个问题：

```
means <- c(0, 1, 2)

output <- double()
```

```
for (i in seq_along(means)) {
  n <- sample(100, 1)
  output <- c(output, rnorm(n, means[[i]]))
}
str(output)
#>  num [1:202] 0.912 0.205 2.584 -0.789 0.588 ...
```

但这并不是一种非常高效的方式，因为 R 要在每次迭代中复制上一次迭代中的所有数据。从技术角度来看，你执行了一种“平方”（$O(n^2)$）操作，这意味着，如果元素数量增加到原来的 3 倍，那么循环时间就要增加到原来的 9 倍。

更好的解决方式是将结果保存在一个列表中，循环结束后再组合成一个向量：

```
out <- vector("list", length(means))
for (i in seq_along(means)) {
  n <- sample(100, 1)
  out[[i]] <- rnorm(n, means[[i]])
}
str(out)
#> List of 3
#>  $ : num [1:83] 0.367 1.13 -0.941 0.218 1.415 ...
#>  $ : num [1:21] -0.485 -0.425 2.937 1.688 1.324 ...
#>  $ : num [1:40] 2.34 1.59 2.93 3.84 1.3 ...
str(unlist(out))
#>  num [1:144] 0.367 1.13 -0.941 0.218 1.415 ...
```

这里我们使用了 `unlist()` 函数将一个向量列表转换为单个向量。更严格的一种转换方式是使用 `purrr::flatten_dbl()` 函数，如果输入不是双精度型列表，那么它就会抛出一个错误。

其他情况下也可以使用这种编码模式。

- 你或许会生成一个很长的字符串。不要使用 `paste()` 函数将每次迭代的结果与上一次连接起来，而应该将每次迭代结果保存在字符向量中，然后再使用 `paste(output, collapse = "")` 将这个字符向量组合成一个字符串。
- 你或许会生成一个很大的数据框。不要在每次迭代中依次使用 `rbind()` 函数，而应该将每次迭代结果保存在列表中，再使用 `dplyr::bind_rows(output)` 将结果组合成数据框。

注意这种模式。只要遇到类似情况，就应该使用一个更复杂的对象来保存每次迭代的结果，最后再一次性组合起来。

16.3.4　未知的序列长度

有时你甚至不知道输入序列的长度。这种情况在模拟时很常见。例如，在掷硬币时，你想要循环到连续 3 次掷出正面向上。这种迭代不能使用 `for` 循环来实现，而应该使用 `while` 循环。`while` 循环比 `for` 循环更简单，因为前者只需要 2 个部分：条件和循环体。

```
while (condition) {
  # 循环体
}
```

`while` 循环也比 `for` 循环更常用，因为任何 `for` 循环都可以使用 `while` 循环重新实现，但不是所有 `while` 循环都能使用 `for` 循环重新实现：

```
for (i in seq_along(x)) {
  # 循环体
}

# 等价于
i <- 1
while (i <= length(x)) {
  # 循环体
  i <- i + 1
}
```

在以下示例中，我们使用 `while` 循环找出了连续 3 次掷出正面向上的硬币所需的投掷次数：

```
flip <- function() sample(c("T", "H"), 1)

flips <- 0
nheads <- 0

while (nheads < 3) {
  if (flip() == "H") {
    nheads <- nheads + 1
  } else {
    nheads <- 0
  }
  flips <- flips + 1
}
flips
#> [1] 3
```

我们只简单介绍了一下 `while` 循环，因为几乎用不到。`while` 循环最常用于模拟，可本书中并不包括这项内容。但是，知道它的存在还是很重要的，当遇到事先不知道迭代次数的问题时，你就可以使用 `while` 循环来解决。

16.3.5 练习

(1) 假设一个目录中全是你想要读入的 CSV 文件。你已经将这些文件的路径保存在向量 `files <- dir("data/", pattern = "\\.csv$", full.names = TRUE)` 中，现在想要使用 `read_csv()` 函数来读取每个文件。编写一个 `for` 循环将这些文件加载到一个数据框中。

(2) 如果使用了 `for (nm in names(x))`，但 x 中并没有名称，那么会发生什么情况？如果 x 中只有部分元素有名称，那么会发生什么情况？如果名称不是唯一的，又会发生什么情况？

(3) 编写一个函数，使其输出一个数据框中所有数值列的均值及名称。例如，`show_mean(iris)` 会输出以下结果。

```
show_mean(iris)
#> Sepal.Length: 5.84
#> Sepal.Width:  3.06
#> Petal.Length: 3.76
#> Petal.Width:  1.20
```

（附加题：虽然变量名称的长度不同，但我们使用了哪个函数确保数值可以整齐排列？）

(4) 以下代码的作用是什么？它是如何运行的？

```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) {
    factor(x, labels = c("auto", "manual"))
  }
)
for (var in names(trans)) {
  mtcars[[var]] <- trans[[var]](mtcars[[var]])
}
```

16.4 for循环与函数式编程

for 循环在 R 中不像在其他语言中那么重要，因为 R 是一门函数式编程语言。这意味着可以先将 for 循环包装在函数中，然后再调用这个函数，而不是直接使用 for 循环。

为了说明函数式编程的重要性，（再次）思考一下这个简单的数据框：

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

假设想要计算每列的均值。你可以使用 for 循环来完成这个任务：

```
output <- vector("double", length(df))
for (i in seq_along(df)) {
  output[[i]] <- mean(df[[i]])
}
output
#> [1]  0.2026 -0.2068 0.1275 -0.0917
```

然后你意识到会非常频繁地计算每列均值，因此将这段代码提取出来，转换成一个函数：

```
col_mean <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}
```

接着你又觉得应该计算出每列的中位数和标准差，因此复制粘贴了函数 col_mean()，并使用 median() 和 sd() 函数替换了 mean() 函数：

```
col_median <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- median(df[[i]])
  }
  output
```

```
}
col_sd <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- sd(df[[i]])
  }
  output
}
```

啊哦！你又复制粘贴了 2 次，因此应该思考一下如何扩展这段代码。注意，这段代码中大部分是一个 for 循环模板，而且很难看出不同函数（mean()、median() 和 sd()）之间的区别。

如果看见以下函数，你应该如何做？

```
f1 <- function(x) abs(x - mean(x)) ^ 1
f2 <- function(x) abs(x - mean(x)) ^ 2
f3 <- function(x) abs(x - mean(x)) ^ 3
```

希望你能够发现这段代码中有很多重复，并可以使用一个附加参数来提取重复代码：

```
f <- function(x, i) abs(x - mean(x)) ^ i
```

你已经减少了出错的概率（因为现在的代码只是原来的 1/3），而且这个函数更容易推广到新场景。

通过添加支持函数应用到每列的一个参数，我们可以使用同一个函数完成与 col_mean()、col_median() 和 col_sd() 函数相同的操作：

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
col_summary(df, median)
#> [1]  0.237 -0.218  0.254 -0.133
col_summary(df, mean)
#> [1]  0.2026 -0.2068  0.1275 -0.0917
```

将函数作为参数传入另一个函数的这种做法是一种非常强大的功能，它是促使 R 成为函数式编程语言的因素之一。你需要花些时间才能真正理解这种思想，但这绝对是值得的。在本章剩余部分，我们将学习和使用 purrr 包，它提供的函数可以替代很多常见的 for 循环应用。R 基础包中的应用函数族（apply()、lapply()、tapply() 等）也可以完成类似的任务，但 purrr 包中的函数更一致，也更易于学习。

使用 purrr 函数代替 for 循环的目的是将常见的列表处理问题分解为独立的几个部分。

- 对于列表中的单个元素，你能找到解决问题的方法吗？如果找到了解决方法，那么你就可以使用 purrr 将这种方法扩展到列表中的所有元素。
- 如果你面临的是一个非常复杂的问题，那么如何将其分解为几个可行的子问题，然后循序渐进地解决，直至完成最终的解决方案？使用 purrr，你可以解决很多子问题，然后再通过管道操作将这些问题的结果组合起来。

这种方法可以让你更轻松地解决新问题，对于老问题，你也可以在重温代码时更轻松地理解当时的解决方案。

练习

(1) 阅读 apply() 函数的文档。在第二种情形中，它会扩展出哪两种 for 循环？

(2) 修改 col_summary() 函数，让其只应用于数值型的列。你可以使用 is_numeric() 函数返回一个逻辑向量，其中的 TRUE 值就对应每个数值型列。

16.5 映射函数

先对向量进行循环、然后对其每个元素进行一番处理，最后保存结果。这种模式太普遍了，因此 purrr 包提供了一个函数族来替你完成这种操作。每种类型的输出都有一个相应的函数：

- map() 用于输出列表；
- map_lgl() 用于输出逻辑型向量；
- map_int() 用于输出整型向量；
- map_dbl() 用于输出双精度型向量；
- map_chr() 用于输出字符型向量。

每个函数都使用一个向量作为输入，并对向量的每个元素应用一个函数，然后返回和输入向量同样长度（同样名称）的一个新向量。向量的类型由映射函数的后缀决定。

一旦掌握了这些函数，你就会发现可以在解决迭代问题时节省大量时间。但你无须因为使用了 for 循环，没有使用映射函数而感到内疚。映射函数是一种高度抽象，需要花费很长时间才能理解其工作原理。重要的事情是解决工作中遇到的问题，而不是写出最简洁优雅的代码（尽管肯定也应该为之努力！）

可能有些人会告诉你不要使用 for 循环，因为它们很慢。这些人完全错了！（至少他们已经赶不上时代了，因为 for 循环已经有很多年都不慢了。）使用 map() 函数的主要优势不是速度，而是简洁：它们可以让你的代码更易编写，也更易读。

我们可以使用这些函数来执行与最后一个 for 循环相同的操作。因为那些摘要函数返回的是双精度数，所以我们需要使用 map_dbl() 函数：

```
map_dbl(df, mean)
#>       a       b      c       d
#>  0.2026 -0.2068 0.1275 -0.0917
map_dbl(df, median)
#>      a      b     c      d
#>  0.237 -0.218 0.254 -0.133
map_dbl(df, sd)
#>     a     b     c     d
#> 0.796 0.759 1.164 1.062
```

与 for 循环相比，映射函数的重点在于需要执行的操作（即 mean()、median() 和 sd()），

而不是在所有元素中循环所需的跟踪记录以及保存结果。如果使用管道，这一点就会表现得更加明显：

```
df %>% map_dbl(mean)
#>       a       b      c       d
#>  0.2026 -0.2068  0.1275 -0.0917
df %>% map_dbl(median)
#>      a      b      c      d
#>  0.237 -0.218  0.254 -0.133
df %>% map_dbl(sd)
#>      a     b     c     d
#>  0.796 0.759 1.164 1.062
```

map_*() 和 col_summary() 具有以下几点区别。

- 所有 purrr 函数都是用 C 实现的。这使得它们的速度非常快，但牺牲了一些可读性。
- 第二个参数（即 .f，要应用的函数）可以是一个公式、一个字符向量或一个整型向量。下一节将介绍这些快捷方式。
- map_*() 使用 ...（参见 14.5.3 节）向 .f 传递一些附加参数，供其在每次调用时使用：

```
map_dbl(df, mean, trim = 0.5)
#>      a      b      c      d
#>  0.237 -0.218  0.254 -0.133
```

- 映射函数还可以保留名称：

```
z <- list(x = 1:3, y = 4:5)
map_int(z, length)
#> x y
#> 3 2
```

16.5.1 快捷方式

对于参数 .f，你可以使用几种快捷方式来减少输入量。假设你想对某个数据集中的每个分组都拟合一个线性模型。以下这个简单示例将 mtcars 数据集拆分成 3 个部分（按照气缸的值分类），并对每个部分拟合一个线性模型：

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(function(df) lm(mpg ~ wt, data = df))
```

因为 R 中创建匿名函数的语法比较繁琐，所以 purrr 提供了一种更方便的快捷方式——单侧公式：

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .))
```

我们在以上示例中使用了 . 作为一个代词：它表示当前列表元素（与 for 循环中用 i 表示当前索引是一样的）。

当检查多个模型时，有时你会需要提取出像 R^2 这样的摘要统计量。要想完成这个任务，

需要先运行 summary() 函数，然后提取出结果中的 r.squared。我们可以使用匿名函数的快捷方式来完成这个操作：

```
models %>%
  map(summary) %>%
  map_dbl(~.$r.squared)
#>     4     6     8
#> 0.509 0.465 0.423
```

因为提取命名成分的这种操作非常普遍，所以 purrr 提供了一种更为简洁的快捷方式：使用字符串。

```
models %>%
  map(summary) %>%
  map_dbl("r.squared")
#>     4     6     8
#> 0.509 0.465 0.423
```

你还可以使用整数按照位置来选取元素：

```
x <- list(list(1, 2, 3), list(4, 5, 6), list(7, 8, 9))
x %>% map_dbl(2)
#> [1] 2 5 8
```

16.5.2 R基础包

如果非常熟悉 R 基础包中的应用函数族，那么你会发现它们与 purrr 函数具有以下共同点。

- lapply() 函数与 map 函数的功能基本相同，差别在于 map() 函数与 purrr 包中的其他函数是一致的，而且可以对 .f 使用快捷方式。
- R 基础包中的 sapply() 函数是对 lapply() 的包装，可以自动简化输出。这对交互工作是有用的，但作为函数则是有问题的，因为你不知道会得到什么样的输出：

```
x1 <- list(
  c(0.27, 0.37, 0.57, 0.91, 0.20),
  c(0.90, 0.94, 0.66, 0.63, 0.06),
  c(0.21, 0.18, 0.69, 0.38, 0.77)
)
x2 <- list(
  c(0.50, 0.72, 0.99, 0.38, 0.78),
  c(0.93, 0.21, 0.65, 0.13, 0.27),
  c(0.39, 0.01, 0.38, 0.87, 0.34)
)

threshold <- function(x, cutoff = 0.8) x[x > cutoff]
x1 %>% sapply(threshold) %>% str()
#> List of 3
#>  $ : num 0.91
#>  $ : num [1:2] 0.9 0.94
#>  $ : num(0)
x2 %>% sapply(threshold) %>% str()
#>  num [1:3] 0.99 0.93 0.87
```

- `vapply()` 函数是 `sapply()` 的一种安全替代方式，因为前者可以提供额外的参数来定义类型。`vapply()` 的唯一缺点是输入量较大：`vapply(df, is.numeric, logical(1))` 等价于 `map_lgl(df, is.numeric)`。`vapply()` 胜于 purrr 中的映射函数的一点是，它可以生成矩阵，而映射函数只能生成向量。

本章重点介绍 purrr 函数，因为它们具有更加一致的名称和参数，还有好用的快捷方式，而且未来还可以轻松实现并行计算，以及方便美观的进度条。

16.5.3　练习

(1) 编写代码以使用一种映射函数完成以下任务。

  a. 计算 `mtcars` 数据集中每列的均值。
  b. 确定 `nycflights13::flights` 数据集中每列的类型。
  c. 计算 `iris` 数据集中每列唯一值的数量。
  d. 分别使用 $\mu = -10$、0、10 和 100 的正态分布生成 10 个随机数。

(2) 如何建立一个向量来表明数据框中的每一列是否为一个因子？

(3) 如果在非列表向量上使用映射函数，那么会发生什么情况？`map(1:5, runif)` 的作用是什么？为什么？

(4) `map(-2:2, rnorm, n = 5)` 的作用是什么？为什么？`map_dbl(-2:2, rnorm, n = 5)` 的作用又是什么？为什么？

(5) 重写 `map(x, function(df) lm(mpg ~ wt, data = df))` 这段代码，去除匿名函数。

16.6　对操作失败的处理

当使用映射函数重复多种操作时，某次操作失败的概率会大大增加。当这种情况发生时，你不仅会收到一条错误消息，而且不会得到任何结果。这很令人恼火：为什么一次失败会使得我们无法得到所有其他成功操作的结果？怎样才能保证不会出现一条鱼腥了一锅汤的情况？

本节将介绍如何使用函数 `safely()` 来处理这种情况。`safely()` 是一个修饰函数（副词），它接受一个函数（动词），对其进行修改并返回修改后的函数。这样一来，修改后的函数就不会抛出错误。相反，它总是会返回由以下两个元素组成的一个列表。

`result`
　原始结果。如果出现错误，那么它就是 `NULL`。

`error`
　错误对象。如果操作成功，那么它就是 `NULL`。

（你可能非常熟悉 R 基础包中的 `try()` 函数，`safely()` 函数确实和它很相似。但因为 `safely()` 有时会返回原始结果，有时会返回错误对象，所以处理起来更困难一些。）

我们使用一个简单的 `log()` 函数来进行说明：

```
safe_log <- safely(log)
str(safe_log(10))
#> List of 2
#>  $ result: num 2.3
#>  $ error : NULL
str(safe_log("a"))
#> List of 2
#>  $ result: NULL
#>  $ error :List of 2
#>   ..$ message: chr "non-numeric argument to mathematical ..."
#>   ..$ call   : language .f(...)
#>   ..- attr(*, "class")= chr [1:3] "simpleError" "error" ...
```

当函数成功运行时，result 元素中包含原始结果，error 元素的值是 NULL；当函数运行失败时，result 元素的值是 NULL，error 元素中包含错误对象。

safely() 也可以与 map() 函数共同使用：

```
x <- list(1, 10, "a")
y <- x %>% map(safely(log))
str(y)
#> List of 3
#>  $ :List of 2
#>   ..$ result: num 0
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: num 2.3
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: NULL
#>   ..$ error :List of 2
#>   .. ..$ message: chr "non-numeric argument to ..."
#>   .. ..$ call   : language .f(...)
#>   .. ..- attr(*, "class")=chr [1:3] "simpleError" "error" ...
```

如果将以上结果转换为两个列表，一个列表包含所有错误对象，另一个列表包含所有原始结果，那么处理起来就会更加容易。可以使用 purrr::transpose() 函数轻松完成这个任务：

```
y <- y %>% transpose()
str(y)
#> List of 2
#>  $ result:List of 3
#>   ..$ : num 0
#>   ..$ : num 2.3
#>   ..$ : NULL
#>  $ error :List of 3
#>   ..$ : NULL
#>   ..$ : NULL
#>   ..$ :List of 2
#>   .. ..$ message: chr "non-numeric argument to ..."
#>   .. ..$ call   : language .f(...)
#>   .. ..- attr(*, "class")=chr [1:3] "simpleError" "error" ...
```

你可以自行决定如何处理错误对象，但一般来说，你应该检查一下 y 中错误对象所对应的 x 值，或者使用 y 中的正常结果进行一些处理：

```
is_ok <- y$error %>% map_lgl(is_null)
x[!is_ok]
#> [[1]]
#> [1] "a"
y$result[is_ok] %>% flatten_dbl()
#> [1] 0.0 2.3
```

purrr 还提供了另外两个有用的修饰函数。

- 与 safely() 类似，possibly() 函数也总是会成功返回。它比 safely() 还要简单一些，因为可以设定出现错误时返回一个默认值：

```
x <- list(1, 10, "a")
x %>% map_dbl(possibly(log, NA_real_))
#> [1] 0.0 2.3  NA
```

- quietly() 函数与 safely() 的作用基本相同，但前者的结果中不包含错误对象，而是包含输出、消息和警告：

```
x <- list(1, -1)
x %>% map(quietly(log)) %>% str()
#> List of 2
#>  $ :List of 4
#>   ..$ result  : num 0
#>   ..$ output  : chr ""
#>   ..$ warnings: chr(0)
#>   ..$ messages: chr(0)
#>  $ :List of 4
#>   ..$ result  : num NaN
#>   ..$ output  : chr ""
#>   ..$ warnings: chr "NaNs produced"
#>   ..$ messages: chr(0)
```

16.7 多参数映射

迄今为止，我们的映射函数都是对单个输入进行映射。但我们经常会有多个相关的输入需要同步迭代，这就是 map2() 和 pmap() 函数的用武之地。例如，假设你想模拟几个均值不同的随机正态分布，我们已经知道了如何使用 map() 函数来完成这个任务：

```
mu <- list(5, 10, -3)
mu %>%
  map(rnorm, n = 5) %>%
  str()
#> List of 3
#>  $ : num [1:5] 5.45 5.5 5.78 6.51 3.18
#>  $ : num [1:5] 10.79 9.03 10.89 10.76 10.65
#>  $ : num [1:5] -3.54 -3.08 -5.01 -3.51 -2.9
```

如果还想让标准差也不同，那么该怎么办呢？其中一种方法是使用均值向量和标准差向量的索引进行迭代：

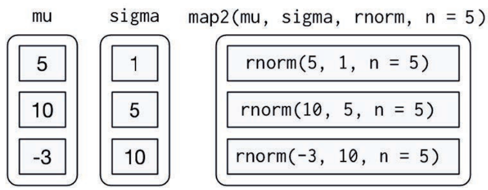
```
sigma <- list(1, 5, 10)
seq_along(mu) %>%
```

```
  map(~rnorm(5, mu[[.]], sigma[[.]])) %>%
  str()
#> List of 3
#>  $ : num [1:5] 4.94 2.57 4.37 4.12 5.29
#>  $ : num [1:5] 11.72 5.32 11.46 10.24 12.22
#>  $ : num [1:5] 3.68 -6.12 22.24 -7.2 10.37
```

但是这种方法很难让人理解代码的本意。相反，我们应该使用 map2() 函数，它可以对两个向量进行同步迭代：

```
map2(mu, sigma, rnorm, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 4.78 5.59 4.93 4.3 4.47
#>  $ : num [1:5] 10.85 10.57 6.02 8.82 15.93
#>  $ : num [1:5] -1.12 7.39 -7.5 -10.09 -2.7
```

map2() 函数可以生成以下一系列函数调用：



注意，每次调用时值发生变化的参数（这里是 mu 和 sigma）要放在映射函数（这里是 rnorm）的**前面**，值保持不变的参数（这里是 n）要放在映射函数的**后面**。

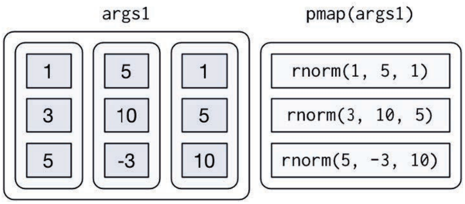
和 map() 函数一样，map2() 函数也是对 for 循环的包装：

```
map2 <- function(x, y, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], y[[i]], ...)
  }
  out
}
```

还可以开发 ma3()、map4()、map5()、map6() 等，但这样你很快就会感到无聊。相反，purrr 提供了 pmap() 函数，它可以将一个列表作为参数。如果你想生成均值、标准差和样本数量都不相同的正态分布，那么就可以使用这个函数：

```
n <- list(1, 3, 5)
args1 <- list(n, mu, sigma)
args1 %>%
  pmap(rnorm) %>%
  str()
#> List of 3
#>  $ : num 4.55
#>  $ : num [1:3] 13.4 18.8 13.2
#>  $ : num [1:5] 0.685 10.801 -11.671 21.363 -2.562
```

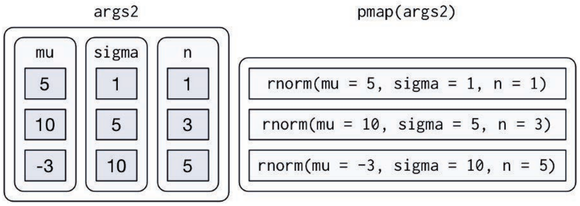
图形表示如下所示：



如果没有为列表的元素命名，那么 pmap() 在调用函数时就会按照位置匹配。这样做比较容易出错，而且会让代码的可读性变差，因此最好使用命名参数：

```
args2 <- list(mean = mu, sd = sigma, n = n)
args2 %>%
  pmap(rnorm) %>%
  str()
```

这样生成的函数调用更长一些，但更安全：



因为长度都是相同的，所以可以将各个参数保存在一个数据框中：

```
params <- tribble(
  ~mean, ~sd, ~n,
   5,     1,  1,
  10,     5,  3,
  -3,    10,  5
)
params %>%
  pmap(rnorm)
#> [[1]]
#> [1] 4.68
#>
#> [[2]]
#> [1] 23.44 12.85  7.28
#>
#> [[3]]
#> [1]  -5.34 -17.66   0.92   6.06   9.02
```

当代码变得比较复杂时，我们认为使用数据框是一种非常好的方法，因为这样可以确保每列都具有名称，而且与其他列具有相同的长度。

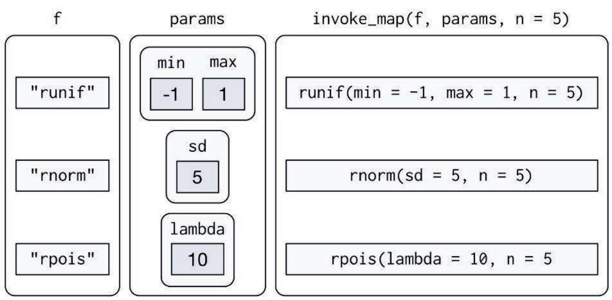
调用不同函数

还有一种更复杂的情况：不但传给函数的参数不同，甚至函数本身也是不同的。

```
f <- c("runif", "rnorm", "rpois")
param <- list(
  list(min = -1, max = 1),
  list(sd = 5),
  list(lambda = 10)
)
```

为了处理这种情况，你可以使用 invoke_map() 函数：

```
invoke_map(f, param, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 0.762 0.36 -0.714 0.531 0.254
#>  $ : num [1:5] 3.07 -3.09 1.1 5.64 9.07
#>  $ : int [1:5] 9 14 8 9 7
```



第一个参数是一个函数列表或包含函数名称的字符向量。第二个参数是列表的一个列表，其中给出了要传给各个函数的不同参数。随后的参数要传给每个函数。

同样地，我们可以通过 tribble() 函数使得这些参数配对更加容易：

```
sim <- tribble(
  ~f,      ~params,
  "runif", list(min = -1, max = 1),
  "rnorm", list(sd = 5),
  "rpois", list(lambda = 10)
)
sim %>%
  mutate(sim = invoke_map(f, params, n = 10))
```

16.8 游走函数

如果调用函数的目的是利用其副作用，而不是返回值时，那么就应该使用游走函数，而不是映射函数。通常来说，使用这个函数的目的是在屏幕上提供输出或者将文件保存到磁盘——重要的是操作过程，而不是返回值。以下是一个非常简单的示例：

```
x <- list(1, "a", 3)

x %>%
  walk(print)
#> [1] 1
#> [1] "a"
#> [1] 3
```

一般来说，walk() 函数不如 walk2() 和 pwalk() 实用。例如，如果有一个图形列表和一个文件名向量，那么你就可以使用 pwalk() 将每个文件保存到相应的磁盘位置：

```
library(ggplot2)
plots <- mtcars %>%
  split(.$cyl) %>%
  map(~ggplot(., aes(mpg, wt)) + geom_point())
paths <- stringr::str_c(names(plots), ".pdf")

pwalk(list(paths, plots), ggsave, path = tempdir())
```

walk()、walk2() 和 pwalk() 都会隐式地返回 .x，即第一个参数。这使得它们非常适用于管道操作。

16.9 for循环的其他模式

purrr 还提供了其他一些函数，可以对 for 循环的其他模式进行抽象。虽然它们的使用频率比映射函数低，但了解一下还是有用的。本节的目的就是对这些函数进行简单介绍，以便你将来遇到类似问题时能够想起它们，再查阅文档以获得更多详细信息。

16.9.1 预测函数

一些函数可以与返回 TRUE 或 FALSE 的**预测函数**一同使用。

keep() 和 discard() 函数可以分别保留输入中预测值为 TRUE 和 FALSE 的元素：

```
iris %>%
  keep(is.factor) %>%
  str()
#> 'data.frame':	150 obs. of  1 variable:
#>  $ Species: Factor w/ 3 levels "setosa","versicolor",..: ...

iris %>%
  discard(is.factor) %>%
  str()
#> 'data.frame':	150 obs. of  4 variables:
#>  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#>  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3 ...
#>  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 ...
#>  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 ...
```

some() 和 every() 函数分别用来确定预测值是否对某个元素为真以及是否对所有元素为真：

```
x <- list(1:5, letters, list(10))

x %>%
  some(is_character)
#> [1] TRUE

x %>%
  every(is_vector)
#> [1] TRUE
```

detect() 函数可以找出预测值为真的第一个元素，detect_index() 函数则可以返回这个元素的位置：

```
x <- sample(10)
x
#>  [1]  8  7  5  6  9  2 10  1  3  4

x %>%
  detect(~ . > 5)
#> [1] 8

x %>%
  detect_index(~ . > 5)
#> [1] 1
```

head_while() 和 tail_while() 分别从向量的开头和结尾找出预测值为真的元素：

```
x %>%
  head_while(~ . > 5)
#> [1] 8 7

x %>%
  tail_while(~ . > 5)
#> integer(0)
```

16.9.2　归约与累计

对于一个复杂的列表，有时你想将其归约为一个简单列表，方式是使用一个函数不断将两个元素合成一个。如果想要将两表间的一个 dplyr 操作应用于多张表，那么这种方法是非常适合的。例如，如果你有一个数据框列表，并想要通过不断将两个数据框连接成一个的方式来最终生成一个数据框：

```
dfs <- list(
  age = tibble(name = "John", age = 30),
  sex = tibble(name = c("John", "Mary"), sex = c("M", "F")),
  trt = tibble(name = "Mary", treatment = "A")
)

dfs %>% reduce(full_join)
#> Joining, by = "name"
#> Joining, by = "name"
#> # A tibble: 2 × 4
#>    name   age   sex treatment
#>   <chr> <dbl> <chr>     <chr>
#> 1  John    30     M      <NA>
#> 2  Mary    NA     F         A
```

或者你想要找出一张向量列表中的向量间的交集：

```
vs <- list(
  c(1, 3, 5, 6, 10),
  c(1, 2, 3, 7, 8, 10),
  c(1, 2, 3, 4, 8, 9, 10)
```

```
)

vs %>% reduce(intersect)
#> [1]  1  3 10
```

reduce() 函数使用一个“二元”函数（即具有两个基本输入的函数），将其不断应用于一个列表，直到最后只剩下一个元素为止。

累计函数与归约函数很相似，但前者会保留所有中间结果。你可以使用它来实现累计求和：

```
x <- sample(10)
x
#>  [1]  6  9  8  5  2  4  7  1 10  3
x %>% accumulate(`+`)
#>  [1]  6 15 23 28 30 34 41 42 52 55
```

16.9.3 练习

(1) 使用 for 循环实现一个自定义的 every() 函数，并将其与 purrr::every() 比较一下。purrr 版的函数具有哪些自定义函数中没有的功能？

(2) 创建一个加强版的 col_sum() 函数，将摘要函数应用于数据框的每个数值列。

(3) R 基础包中可能与 col_sum() 等价的一个函数是：

```
col_sum3 <- function(df, f) {
  is_num <- sapply(df, is.numeric)
  df_num <- df[, is_num]

  sapply(df_num, f)
}
```

但这个函数在使用以下输入时出现了几个 bug：

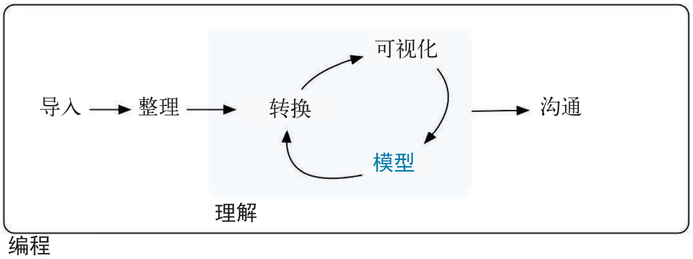
```
df <- tibble(
  x = 1:3,
  y = 3:1,
  z = c("a", "b", "c")
)
# 没问题
col_sum3(df, mean)
# 有问题：不要总是返回数字向量
col_sum3(df[1:2], mean)
col_sum3(df[1], mean)
col_sum3(df[0], mean)
```

引发 bug 的原因是什么呢？

第四部分

模型

我们已经掌握了强大的编程工具，现在终于可以开始建模了。在这一部分中，我们将使用新的数据处理和编程工具来拟合多种模型，并理解其工作原理。本书的重点在于数据探索，而不是假设验证或正式推断，但你还是可以学到几种基本工具，以帮助你理解模型中的变化。



模型的作用是提供一个简单的、低维度的数据集摘要。理想情况下，模型可以捕获真正的“信号”（即由我们感兴趣的现象生成的模式），并忽略“噪声”（即我们不感兴趣的随机变动）。这里我们只介绍“预测”模型，顾名思义，也就是能够生成预测的模型。我们不打算在本书中讨论另一类模型，即“数据发现”模型。这种模型的目标不是进行预测，而是帮助我们发现数据中有趣的关系。（这两类模型有时分别称为监督式模型和非监督式模型，但我们认为这两个术语不是很有启发性。）

本书不会使得你对模型背后的数学理论有更深刻的理解，但会帮助你建立对统计模型工作原理的直观认识，并且会向你介绍一整套有用的工具，以便帮助你使用模型来更深刻地理解数据。

- 第 17 章将介绍模型的运行机制，重点在于一些重要的线性模型。你将掌握一些通用工具，以深刻理解预测模型如何对数据进行预测。我们将主要使用简单的模拟数据集进行介绍。

- 第 18 章将介绍如何使用模型从真实数据中提取已知模式。一旦识别出一种重要模式，那你就应该用一个模型将其明确表示出来，因为随后你就可以更容易地发现剩余的微妙信号。
- 第 19 章将介绍如何使用多个简单模型来帮助理解复杂数据集。这是一种非常强大的技术，但需要结合使用建模工具和编程工具才能掌握这种技术。

这些内容都非常重要，因为我们没有对定量评估模型工具做任何介绍。我们这样做是经过深思熟虑的：精确地量化模型需要大量背景知识，而本书无法一一介绍。现在，你能依靠的只有定性评估技术和自己的怀疑精神。18.4 节会介绍一些其他资源，以供你进一步学习。

假设生成和假设验证

在本书中，我们将模型作为一种数据探索工具，这样就补全了第一部分中介绍过的 EDA 三工具。这不是常用的模型学习方式，但正如你将看到的，模型确实是一种重要的数据探索工具。通常来说，建模的重点在于推断或验证假设是否为真。正确地完成这些任务并不复杂，但相当困难。为了正确进行推断，你必须明确以下两点。

- 每个观测都可以用于数据探索，也可以用于假设验证，但不能同时在二者中使用。
- 在进行数据探索时，一个观测可以使用任意多次，但进行假设验证时，一个观测只能使用一次。一旦使用两次观测，假设验证就会变成数据探索。

这两点是非常必要的，因为要想验证假设，你必须使用与生成假设的数据无关的数据。否则，你就过于乐观了。在进行数据探索时，没有完全错误的结论，但永远不能将探索性分析与验证性分析混为一谈，因为这样做肯定会让你误入歧途。

如果想要严肃认真地进行验证性分析，一种方法是在进行分析前将数据分成 3 个部分。

- 将 60% 的数据作为**训练集**，或称**探索集**。你可以对这部分数据进行任意操作，比如可视化，或者用数据拟合多个模型。
- 将 20% 的数据作为**查询集**。你可以使用这部分数据来比较模型或者进行手动可视化，但不能将其用于自动化过程。
- 将 20% 的数据留作**测试集**。这部分数据**只能使用一次**，用于测试最终模型。

通过这种数据划分方法，你可以使用训练集数据进行探索，偶尔生成候选假设，并使用查询集数据进行验证。确信已经得到正确的模型后，你就可以使用测试集数据进行一次验证了。

（注意，即使正在进行验证性建模，你还是需要做 EDA。如果不做任何 EDA，那么你就会对数据的质量问题一无所知。）

第 17 章

使用modelr实现基础模型

17.1 简介

建立模型的目的是提供一个简单的、低维度的数据集摘要。在本书中，我们使用模型的目的是将数据划分为模式和残差。因为强大的模式往往会掩盖住微妙的趋势，所以我们将借助模型探索数据集，一层层地剥开覆盖在数据集结构上的神秘面纱。

但是，在开始对感兴趣的真实数据应用模型前，我们需要先理解模型的工作原理。因此，本章是特殊的一章，它使用的是模拟数据集。这些数据集非常简单，而且一点儿也不有趣，但它们确实可以帮助你理解建模的本质，这样你就可以在下一章中使用同样的技术来处理真实数据了。

建模过程可以分为两个阶段。

(1) 首先，你需要定义一个**模型族**来表示一种精确但一般性的模式，这种模式就是我们想要捕获的。例如，模式可以是一条直线或一条二次曲线。你可以用方程来表示模型族，比如 `y = a_1 * x + a_2` 或 `y = a_1 * x ^ a_2`，其中 `x` 和 `y` 是数据集中的已知变量，`a_1` 和 `a_2` 是参数。我们可以通过改变参数来捕获不同的模式。

(2) 接下来你要生成一个**拟合模型**，方法是从模型族中找出最接近数据的一个模型。这个阶段使得一般性的模型族具体化为特定模型，如 `y = 3 * x + 7` 或 `y = 9 * x ^ 2`。

拟合模型只是模型族中与数据最接近的一个模型，理解这一点非常重要。这意味着你找到了“最佳”模型（按照某些标准），但并不意味着你找到了良好的模型，而且也绝不代表这个模型是“真的”。George Box 有一句名言说得很好：

> 所有模型都是错误的，但有些是有用的。

这句名言的完整上下文值得一读。

> 如果一个简单模型可以精确表示真实世界中的某个系统，那将非常了不起。然而，巧妙地选择简约模型经常可以提供非常好的近似表示。例如，定律 $pV = RT$ 通过一个常数 R 将“理想”气体的压强 p、体积 V 和温度 T 关联起来。虽然这对于任何真实气体来说都是不精确的，但在很多情况下都是一种良好的近似。此外，这个方程的结构包含非常丰富的信息，因为它来自于对气体分子行为的实证研究。
>
> 对于这样的模型，我们不需要提出“这个模型是真的吗？”这类问题。如果“真”的含义是“绝对真”，那么答案肯定是“不”。我们唯一感兴趣的问题是：“模型是否具有启发性，是否有用？”

模型的目标不是发现真理，而是获得简单但有价值的近似。

准备工作

本章将使用 modelr 包对 R 基础包中的建模函数进行包装，使其可以支持管道操作。

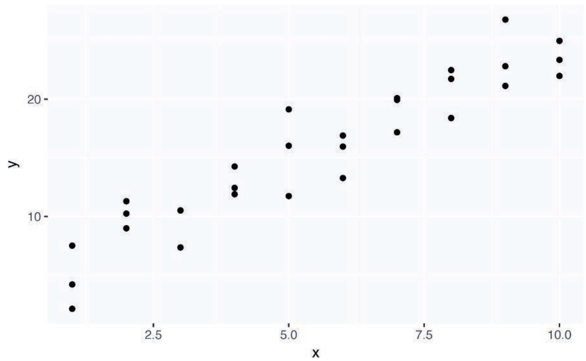
```
library(tidyverse)

library(modelr)
options(na.action = na.warn)
```

17.2　一个简单模型

我们研究一下模拟数据集 sim1，它包含两个连续型变量 x 和 y。我们将这两个变量绘制出来，以查看二者间的关系：

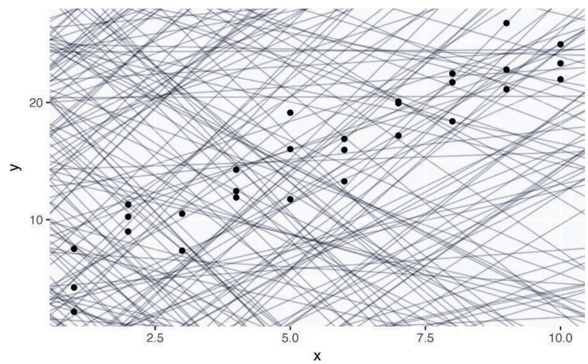
```
ggplot(sim1, aes(x, y)) +
  geom_point()
```



你可以看到数据中存在一种非常强的模式。接下来我们使用模型来捕获这种模式，并将其明确表示出来。我们的任务是确定模型的基本形式。在以上示例中，变量间的关系应该是线性的，即 `y= a_0 + a_1 * x`。首先，我们随机生成一些模型，并将其覆盖到数据上，通过这种方式感受一下这个模型族中的模型形式。对于这个简单示例，我们可以使用 `geom_abline()` 函数，它接受斜率和截距作为参数。随后我们将学习可以用于任意模型的更通用的技术：

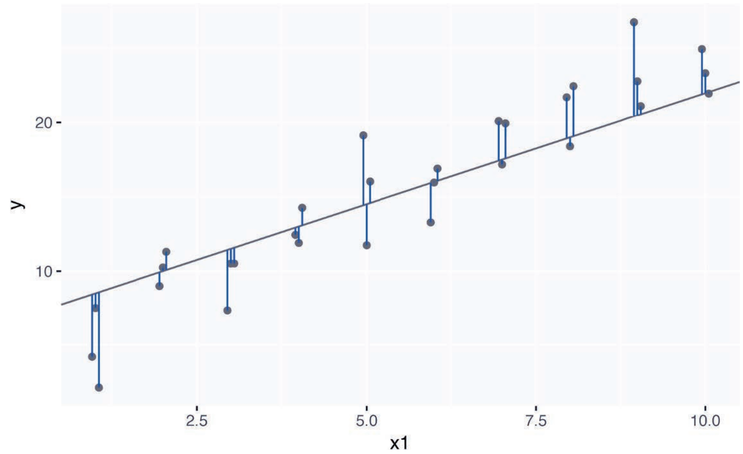
```
models <- tibble(
  a1 = runif(250, -20, 40),
  a2 = runif(250, -5, 5)
)

ggplot(sim1, aes(x, y)) +
  geom_abline(
    aes(intercept = a1, slope = a2),
    data = models, alpha = 1/4
  ) +
  geom_point()
```



这张图中有 250 个模型，但很多都是非常糟糕的！直觉告诉我们，良好的模型应该与数据非常“接近”，因此我们需要一种方法来量化数据与模型之间的距离。然后，找出使得模型与数据间的距离最近的 a_0 和 a_1 的值，就可以拟合出最优模型。

其中一种简单的方法是找出每个数据点与模型之间的垂直距离，如下图所示。（注意，我们将 x 值稍稍移动了一下，以便能够看清每个距离。）



这个距离就是由模型计算出的 y 值（**预测值**）与数据中的实际 y 值（**响应变量**）之间的差。

为了计算出这个距离，首先要将模型族转换为一个 R 函数。这个函数将模型参数和数据作

为输入，并使用模型预测值作为输出：

```
model1 <- function(a, data) {
  a[1] + data$x * a[2]
}
model1(c(7, 1.5), sim1)
#>  [1]  8.5  8.5  8.5 10.0 10.0 10.0 11.5 11.5 11.5 13.0 13.0
#> [12] 13.0 14.5 14.5 14.5 16.0 16.0 16.0 17.5 17.5 17.5 19.0
#> [23] 19.0 19.0 20.5 20.5 20.5 22.0 22.0 22.0
```

接下来，我们需要某种方法来计算预测值与实际值之间的总体距离。换句话说，图中显示了 30 个距离，我们如何将这些距离转换成一个数值呢？

要想完成这个任务，统计学中的一种常用方法是计算“均方根误差”。先计算实际值与预测值之间的差，对其取平方，然后求平均数，最后再计算出平方根。这种距离表示方式具有很多奇妙的数学特性，这里就不介绍了。相信我们吧，没错的！

```
measure_distance <- function(mod, data) {
  diff <- data$y - model1(mod, data)
  sqrt(mean(diff ^ 2))
}
measure_distance(c(7, 1.5), sim1)
#> [1] 2.67
```

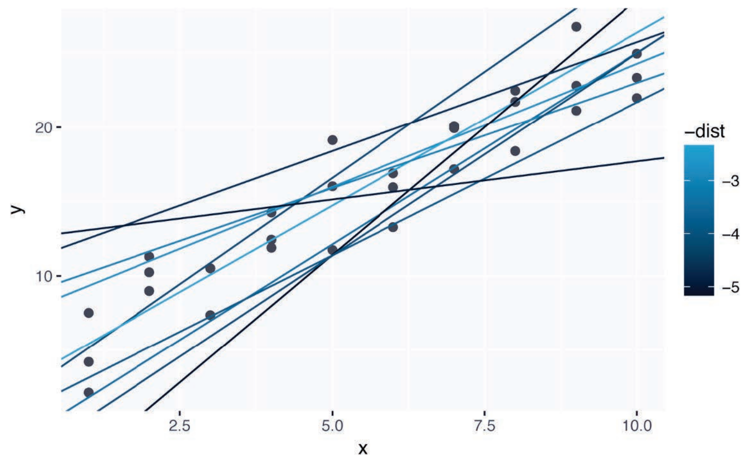
现在可以使用 purrr 来计算前面定义的所有模型和数据间的距离了。我们需要一个辅助函数，因为距离函数希望模型是一个长度为 2 的数值向量：

```
sim1_dist <- function(a1, a2) {
  measure_distance(c(a1, a2), sim1)
}

models <- models %>%
  mutate(dist = purrr::map2_dbl(a1, a2, sim1_dist))
models
#> # A tibble: 250 × 3
#>       a1      a2  dist
#>    <dbl>   <dbl> <dbl>
#> 1 -15.15  0.0889  30.8
#> 2  30.06 -0.8274  13.2
#> 3  16.05  2.2695  13.2
#> 4 -10.57  1.3769  18.7
#> 5 -19.56 -1.0359  41.8
#> 6   7.98  4.5948  19.3
#> # ... with 244 more rows
```

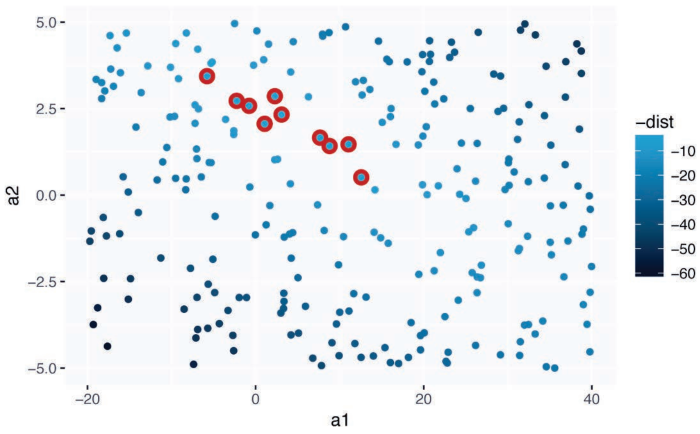
下一步，我们将最好的 10 个模型覆盖到数据上。使用 -dist 为模型上色，这样我们就很容易看出，最佳模型（即距离最小的模型）具有最明亮的颜色：

```
ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, color = "grey30") +
  geom_abline(
    aes(intercept = a1, slope = a2, color = -dist),
    data = filter(models, rank(dist) <= 10)
  )
```



我们还可以将这些模型看作观测，并使用由 a1 和 a2 组成的一张散点图来表示它们，还是使用 -dist 进行上色。虽然这样不能直接看到模型与数据间的比较，但我们可以同时看到很多模型。同样，我们高亮显示前 10 个最佳模型，这次是通过在其下面画出红色圆圈：

```
ggplot(models, aes(a1, a2)) +
  geom_point(
    data = filter(models, rank(dist) <= 10),
    size = 4, color = "red"
  ) +
  geom_point(aes(colour = -dist))
```

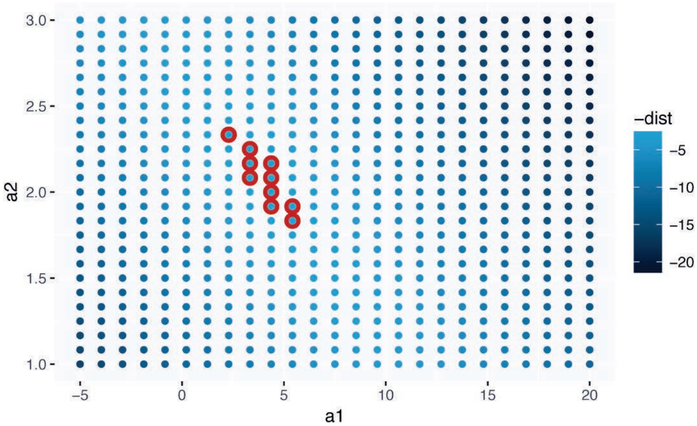


相较于检查多个随机模型，我们使用一种更加系统化的方法来找出模型参数（这种方法称为网格搜索法）。首先，我们生成一张分布均匀的数据点网格，然后将这个网格与前面图中的 10 个最佳模型绘制在一张图中，凭借最佳模型在网格中的位置就可以找出模型参数的粗略值：

```
grid <- expand.grid(
  a1 = seq(-5, 20, length = 25),
  a2 = seq(1, 3, length = 25)
```

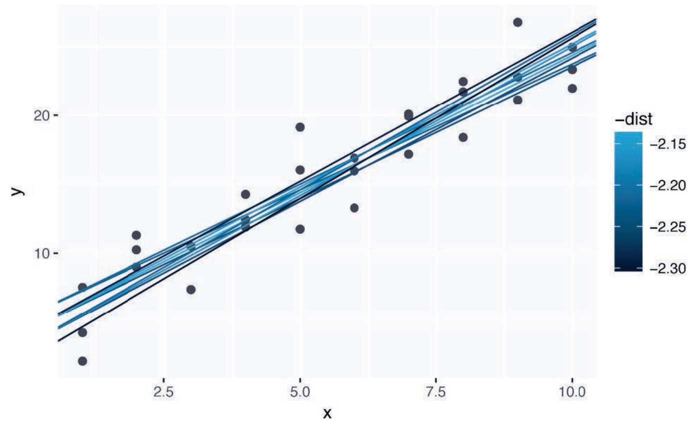
```
  ) %>%
  mutate(dist = purrr::map2_dbl(a1, a2, sim1_dist))

grid %>%
  ggplot(aes(a1, a2)) +
  geom_point(
    data = filter(grid, rank(dist) <= 10),
    size = 4, colour = "red"
  ) +
  geom_point(aes(color = -dist))
```



如果将这 10 个最佳模型重新覆盖到原始数据上，就可以看出效果还是很不错的：

```
ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, color = "grey30") +
  geom_abline(
    aes(intercept = a1, slope = a2, color = -dist),
    data = filter(grid, rank(dist) <= 10)
  )
```

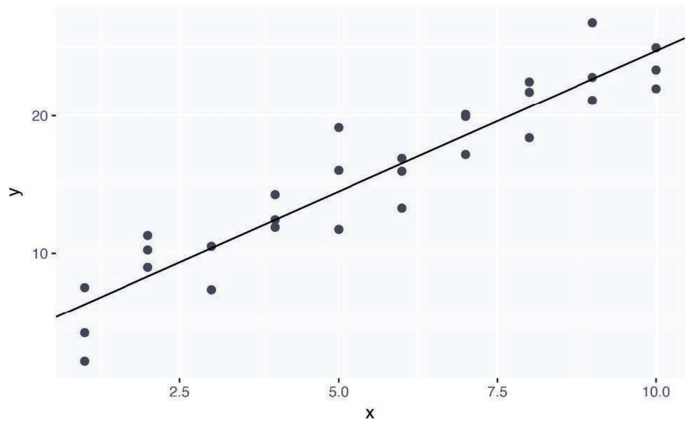


可以设想不断细化网格来最终找出最佳模型。但还有一个更好的方法可以解决这个问题，这种方法是名为“牛顿—拉夫逊搜索”的数值最小化工具。牛顿—拉夫逊方法的直观解释

非常简单：先选择一个起点，环顾四周找到最陡的斜坡，并沿着这个斜坡向下滑行一小段，然后不断重复这个过程，直到不能再下滑为止。在 R 中，我们可以使用 `optim()` 函数来完成这个任务：

```
best <- optim(c(0, 0), measure_distance, data = sim1)
best$par
#> [1] 4.22 2.05

ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, color = "grey30") +
  geom_abline(intercept = best$par[1], slope = best$par[2])
```



无须过多担心 `optim()` 函数的工作细节。现在重要的是要建立直觉。如果具有定义模型与数据集间距离的函数，以及可以通过修改模型参数使距离最小化的算法，那么我们就可以找出最佳模型。以上这种方法的好处是，只要能够写出方程来表示模型族，那么你就可以使用这种方法。

对于这个模型，我们还可以使用另一种方法，因为它是一个更广泛模型族的一种特殊情况，即是线性模型。线性模型的一般形式是 `y = a_1 + a_2 * x_1 + a_3 * x_2 + ... + a_n * x_(n - 1)`。因此，这个简单模型就等价于 n 为 2 且 `x_1` 为 x 的一般线性模型。R 中有专门用于拟合线性模型的工具，即 `lm()` 函数。`lm()` 用一种特殊方法来表示模型族：公式。公式的形式为 `y ~ x`，`lm()` 会将这种形式的公式转换成类似 `y = a_1 + a_2 * x` 的函数。我们使用 `lm()` 来拟合这个模型，并检查输出：

```
sim1_mod <- lm(y ~ x, data = sim1)
coef(sim1_mod)
#> (Intercept)           x
#>        4.22        2.05
```

这与我们使用 `optim()` 函数得到的结果完全一致。`lm()` 函数使用的并不是 `optim()`，而是利用了线性模型的数学结构。实际上，`lm()` 使用了一种非常复杂的算法，通过几何学、微积分和线性代数间的一些关系，它只需要一个步骤就可以找出最近似的模型。这种方法的速度非常快，而且一定能找到全局最小值。

练习

(1) 线性模型的一个缺点是，对异常值非常敏感，因为距离是用平方项表示的。使用以下模拟数据拟合一个线性模型，并对结果进行可视化表示。重新运行几次代码，以生成不同的模拟数据集。对于这个模型，你有什么发现？

```
sim1a <- tibble(
  x = rep(1:10, each = 3),
  y = x * 1.5 + 6 + rt(length(x), df = 2)
)
```

(2) 要想使得线性模型更加健壮，其中一种方法是使用另一种距离度量方式。例如，除了使用均方根误差，你还可以使用平均绝对值距离：

```
measure_distance <- function(mod, data) {
  diff <- data$y - make_prediction(mod, data)
  mean(abs(diff))
}
```

使用 `optim()` 函数与前面的模拟数据拟合模型，并与前面的线性模型对比一下。

(3) 使用数值最优化算法的一个问题是，只能保证得到局部最优值。对于以下的 3 参数模型，执行最优化时会有什么问题？

```
model1 <- function(a, data) {
  a[1] + data$x * a[2] + a[3]
}
```

17.3 模型可视化

对于简单的模型，比如上一节中的模型，我们可以通过仔细检查模型族和拟合系数来找出模型捕获的模式。如果学过关于建模的统计学知识，那么你就可以花费大量时间来做这件事。但这里我们准备介绍另外一种方法，即重点通过预测来理解模型。这种方法的一大优点是，每种类型的预测模型都要进行预测（否则还有什么用处？），因此我们可以使用同样的技术来理解任何类型的预测模型。

找出模型未捕获的信息也是非常有用的，即所谓的残差，它是数据去除预测值后剩余的部分。残差是非常强大的，因为它允许我们使用模型去除数据中显著的模式，以便对剩余的微妙趋势进行研究。

17.3.1 预测

要想对模型的预测进行可视化表示，首先要生成一个分布均匀的数值网格，以覆盖数据所在区域。完成这个任务最简单的方式就是使用 `modelr::data_grid()` 函数，其第一个参数是一个数据框，对于随后的每个参数，它都会找出其中的唯一值，然后生成所有组合：

```
grid <- sim1 %>%
  data_grid(x)
grid
```

```
#> # A tibble: 10 × 1
#>       x
#>   <int>
#> 1     1
#> 2     2
#> 3     3
#> 4     4
#> 5     5
#> 6     6
#> # ... with 4 more rows
```

（如果向模型中添加更多变量，那么结果会更有趣。）

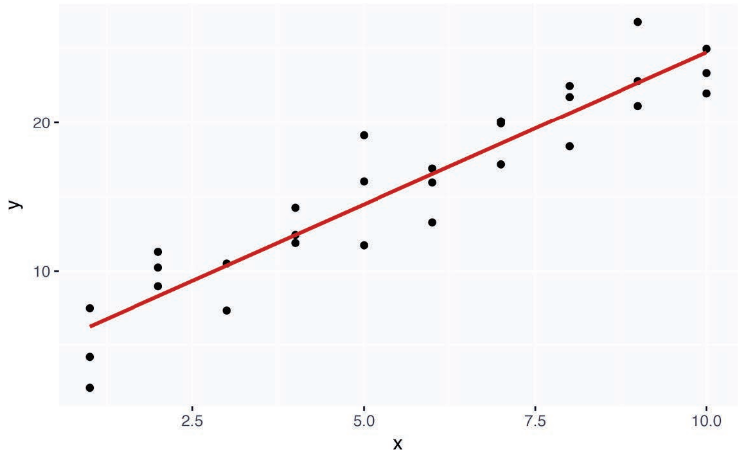
接着添加预测值。我们使用的是 `modelr::add_predictions()` 函数，其参数是一个数据框和一个模型。这个函数可以将模型的预测值作为一个新列添加到数据框中：

```
grid <- grid %>%
  add_predictions(sim1_mod)
grid
#> # A tibble: 10 × 2
#>       x  pred
#>   <int> <dbl>
#> 1     1  6.27
#> 2     2  8.32
#> 3     3 10.38
#> 4     4 12.43
#> 5     5 14.48
#> 6     6 16.53
#> # ... with 4 more rows
```

（你也可以使用这个函数将预测值添加到原始数据集中。）

下一步是绘制预测值。你可能很奇怪，为什么我们要做这些额外的工作，而不是直接使用 `geom_abline()` 函数。因为这种方法适合 R 中的**所有**模型，不管是最简单的还是最复杂的，它只受可视化能力的限制。如果想要对更加复杂的模型进行可视化，可以参考 http://vita.had.co.nz/papers/model-vis.html 来获取更多相关信息。

```
ggplot(sim1, aes(x)) +
  geom_point(aes(y = y)) +
  geom_line(
    aes(y = pred),
    data = grid,
    colour = "red",
    size = 1
  )
```



17.3.2 残差

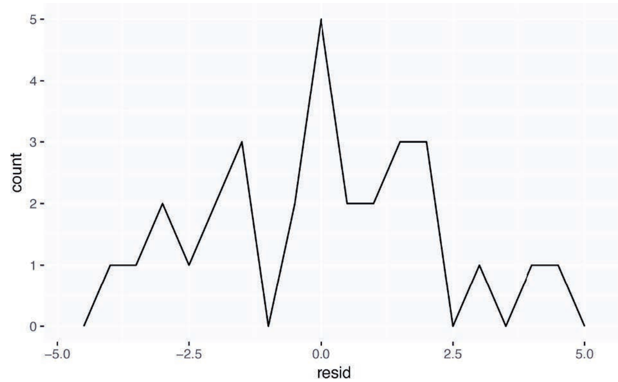
与预测值相对的是**残差**。预测值可以告诉我们模型捕获的模式，残差则表示模型漏掉的部分。残差就是我们前面计算过的观测值与预测值间的距离。

我们可以使用 `add_residuals()` 函数将残差添加到数据中，这个函数与 `add_predictions()` 非常相似。但注意，我们使用的是原始数据，不是生成的网格，因为计算残差需要使用实际的 y 值：

```
sim1 <- sim1 %>%
  add_residuals(sim1_mod)
sim1
#> # A tibble: 30 × 3
#>       x     y  resid
#>   <int> <dbl>  <dbl>
#> 1     1  4.20 -2.072
#> 2     1  7.51  1.238
#> 3     1  2.13 -4.147
#> 4     2  8.99  0.665
#> 5     2 10.24  1.919
#> 6     2 11.30  2.973
#> # ... with 24 more rows
```

对于残差可以反映出模型的哪些信息，有几种不同的理解方法。其中一种方法是简单地绘制频率多边形图，以帮助我们理解残差的分布：

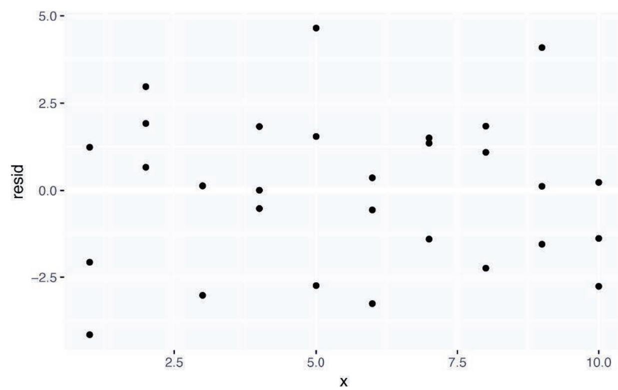
```
ggplot(sim1, aes(resid)) +
  geom_freqpoly(binwidth = 0.5)
```



这种方法可以反映出模型的质量：模型预测值与实际观测值的差别有多大？注意，残差的平均值总是为 0。

我们经常会使用残差代替原来的预测变量来重新绘图。你将在下一章中看到大量这种操作：

```
ggplot(sim1, aes(x, resid)) +
  geom_ref_line(h = 0) +
  geom_point()
```



由上图可知，残差应该是随机的噪声，这表明我们的模型非常好地捕获了数据集中的模式。

17.3.3 练习

(1) 除了使用 lm() 函数拟合一条直线，你还可以使用 loess() 函数来拟合一条平滑曲线。使用 loess() 代替 lm() 对 sim1 数据集重复模型拟合、网格生成、预测和可视化的过程，并将结果与 geom_smooth() 函数进行比较。

(2) add_predictions() 函数还伴有 2 个函数：gather_predictions() 和 spread_predicitons()。这 3 个函数有什么不同？

(3) geom_ref_line() 函数的功能是什么？它来自于哪个 R 包？在显示残差的图形中显示一条参考线是非常重要和有用的，为什么这么说呢？

(4) 为什么需要检查残差绝对值的频率多边形图？与检查残差本身相比，这种方式有什么优缺点呢？

17.4 公式和模型族

在前面使用 `facet_wrap()` 和 `facet_grid()` 函数时，我们已经见过了公式。在 R 中，公式是表示"特殊行为"的一种通用方式。公式不对变量立刻进行求值，只是将变量表示为函数能够理解的形式。

R 中的绝大多数建模函数都使用一种标准转换将公式转换为表示模型族的方程。我们已经见过了一个简单的转换：`y ~ x` 转换为 `y = a_1 + a_2 * x`。如果想看 R 到底进行了什么转换，可以使用 `model_matrix()` 函数。这个函数接受一个数据框和一个公式，并返回一个定义了模型方程的 tibble，其中每一列都关联到方程的一个系数，方程的形式总是类似于 `y = a_1 * out1 + a_2 * out_2`。对于最简单的情况，`y ~ x1`，这个函数会返回以下有趣的结果：

```
df <- tribble(
  ~y, ~x1, ~x2,
  4, 2, 5,
  5, 1, 6
)
model_matrix(df, y ~ x1)
#> # A tibble: 2 × 2
#>   `(Intercept)`    x1
#>           <dbl> <dbl>
#> 1             1     2
#> 2             1     1
```

R 向模型加入截距项的方法是，加入一个值全是 1 的列。默认情况下，R 总是加入这一列。如果不想要截距项，那么你必须使用 `-1` 来明确丢弃它：

```
model_matrix(df, y ~ x1 - 1)
#> # A tibble: 2 × 1
#>      x1
#>   <dbl>
#> 1     2
#> 2     1
```

如果向模型中添加更多变量，那么模型矩阵当然也会随之增长：

```
model_matrix(df, y ~ x1 + x2)
#> # A tibble: 2 × 3
#>   `(Intercept)`    x1    x2
#>           <dbl> <dbl> <dbl>
#> 1             1     2     5
#> 2             1     1     6
```

这种公式表示法有时也称为"Wilkinson-Rogers 表示法"，由 G. N. Wilkinson 和 C. E. Rogers 在其论文"Symbolic Description of Factorial Models for Analysis of Variance"中首次提出。如果想要了解这种模型表示法的全部细节，可以仔细研究一下这篇论文的原文。

以下各节展开介绍了如何使用这种公式表示法来表示分类变量、交互项以及变量转换。

17.4.1 分类变量

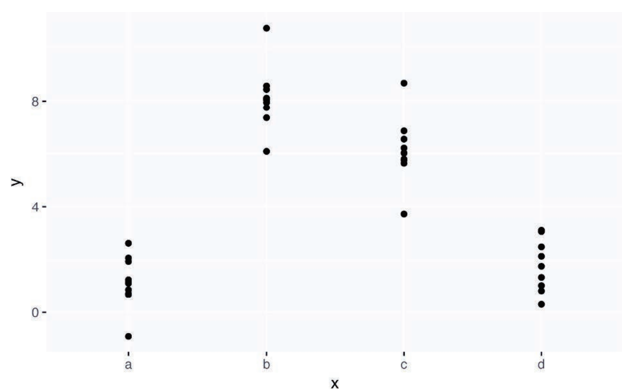
如果预测变量是连续的，那么从公式转换为方程是很简单的；但当预测变量是分类变量时，事情就有点复杂了。假设存在公式 `y ~ sex`，其中 `sex` 的值要么是男性要么是女性，那么将其转换成 `y = x_0 + x_1 * sex` 就没有意义了，因为 `sex` 不是数值，我们不能对它使用乘法。相反，R 的做法是将其转换为 `y = x_0 + x_1 * sex_male`，如果 `sex` 为男性，那么 `sex_male` 的值就是 1，否则其值就是 0：

```
df <- tribble(
  ~ sex, ~ response,
  "male", 1,
  "female", 2,
  "male", 1
)
model_matrix(df, response ~ sex)
#> # A tibble: 3 × 2
#>   `(Intercept)` sexmale
#>           <dbl>   <dbl>
#> 1             1       1
#> 2             1       0
#> 3             1       1
```

你可能想知道，为什么 R 没有同时建立一个 `sexfemale` 列。问题是，这样创建出来的列是可以基于其他列完美预测的（即 `sexfemale = 1 - sexmale`）。可惜的是，这构成问题的具体原因已经超出了本书的讨论范围，大致就是这样做会建立一个过于灵活的模型族，从而生成对数据拟合效果相同的无数个模型。

好在如果重点关注对预测值的可视化，那么就无须担心具体的参数值。我们使用数据和模型进行具体的说明。以下是 modelr 中的 `sim2` 数据集：

```
ggplot(sim2) +
  geom_point(aes(x, y))
```



我们可以拟合一个模型，并生成预测：

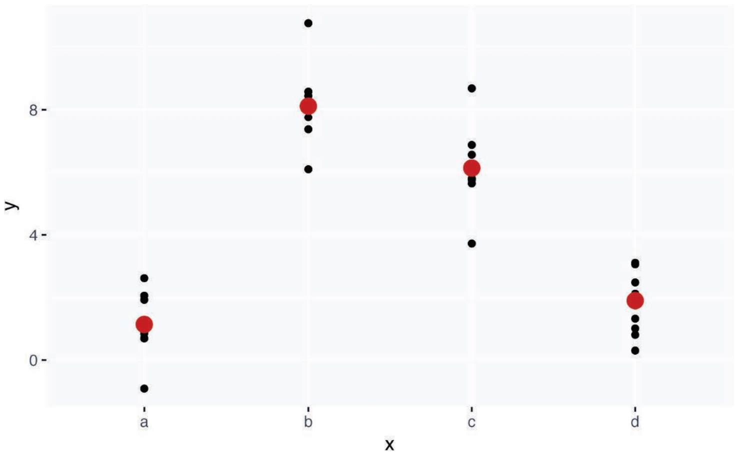
```
mod2 <- lm(y ~ x, data = sim2)

grid <- sim2 %>%
```

```
  data_grid(x) %>%
  add_predictions(mod2)
grid
#> # A tibble: 4 × 2
#>       x  pred
#>   <chr> <dbl>
#> 1     a  1.15
#> 2     b  8.12
#> 3     c  6.13
#> 4     d  1.91
```

实际上，带有分类变量 x 的模型会为每个分类预测出均值。（为什么？因为均值会使均方根距离最小化。）如果将预测值覆盖到原始数据上，就很容易看出这一点：

```
ggplot(sim2, aes(x)) +
  geom_point(aes(y = y)) +
  geom_point(
    data = grid,
    aes(y = pred),
    color = "red",
    size = 4
  )
```



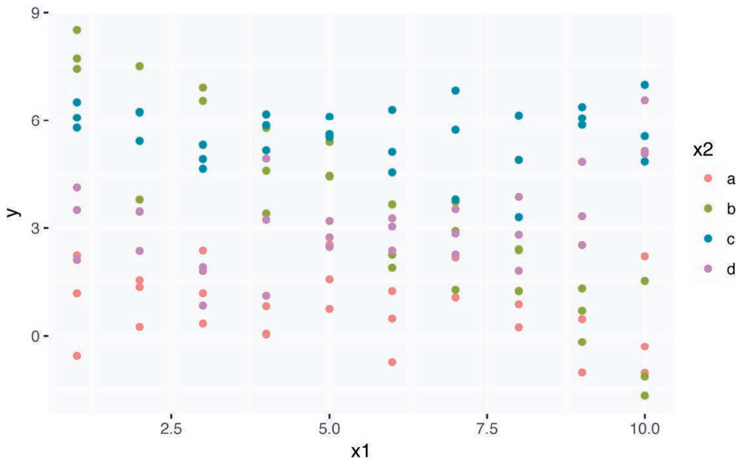
不能对未观测到的水平进行预测。因为有时会不小心进行这种预测，所以我们应该看懂以下这条错误信息：

```
tibble(x = "e") %>%
  add_predictions(mod2)
#> Error in model.frame.default(Terms, newdata, na.action =
#> na.action, xlev = object$xlevels): factor x has new level e
```

17.4.2　交互项（连续变量与分类变量）

如果将一个连续变量与一个分类变量组合使用，会是什么情况？ sim3 数据集中包含了一个分类预测变量和一个连续预测变量，我们可以使用一张简单的图来表示它们：

```
ggplot(sim3, aes(x1, y)) +
  geom_point(aes(color = x2))
```



你可以使用两种模型来拟合这份数据：

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

如果使用 + 添加变量，那么模型会独立地估计每个变量的效果，不考虑其他变量。如果使用 `*`，那么拟合的就是所谓的交互项。例如，`y ~ x1 * x2` 会转换为 `y = a_0 + a_1 * x1 + a_2 * x2 + a_12 * x1 * x2`。注意，只要使用了 `*`，交互项及其各个组成部分就都要包括在模型中。

要想对这样的模型进行可视化，需要两种新技巧。

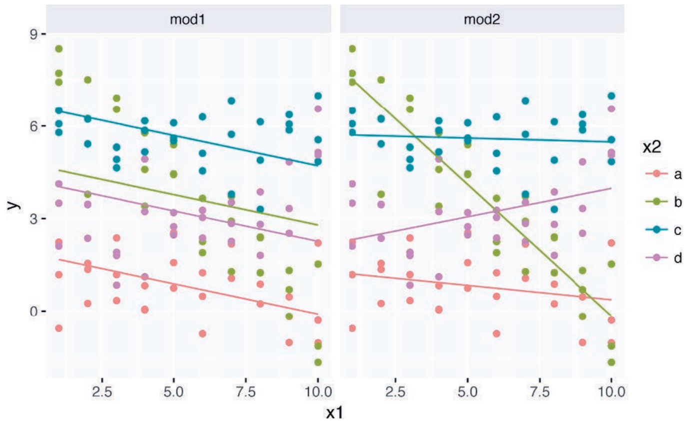
- 因为有两个预测变量，所以我们需要将这两个变量都传给 `data_grid()` 函数。这个函数会找出 `x1` 和 `x2` 中的所有唯一值，并生成所有组合。
- 要想为以上的两个模型同时生成预测，可以使用 `gather_predictions()` 函数，它可以将每个预测作为一行加入数据框。与 `gather_predictions()` 互补的函数是 `spread_predictions()`，后者可以将每个预测作为一列加入数据框。

综上所述，可以得到：

```
grid <- sim3 %>%
  data_grid(x1, x2) %>%
  gather_predictions(mod1, mod2)
grid
#> # A tibble: 80 × 4
#>   model    x1     x2  pred
#>   <chr> <int> <fctr> <dbl>
#> 1  mod1     1      a  1.67
#> 2  mod1     1      b  4.56
#> 3  mod1     1      c  6.48
#> 4  mod1     1      d  4.03
#> 5  mod1     2      a  1.48
#> 6  mod1     2      b  4.37
#> # ... with 74 more rows
```

我们可以使用分面技术将两个模型的可视化结果放在一张图中：

```
ggplot(sim3, aes(x1, y, color = x2)) +
  geom_point() +
  geom_line(data = grid, aes(y = pred)) +
  facet_wrap(~ model)
```

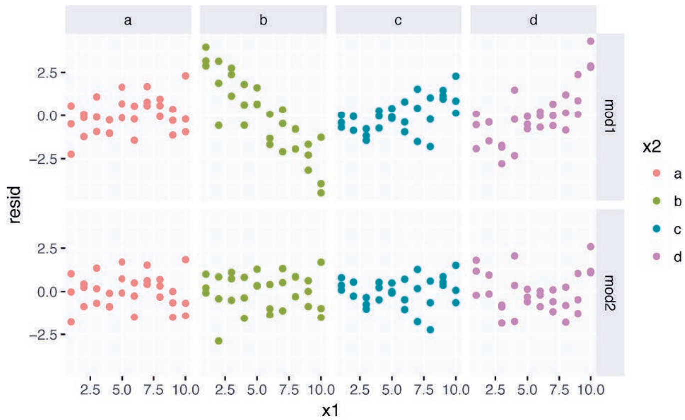


注意，在使用 + 的模型中，每条直线都具有同样的斜率，但截距不同。在使用 * 的模型中，每条直线的斜率和截距都不相同。

对于这份数据来说，哪种模型更好呢？我们可以检查一下残差。这里我们使用了模型和 x2 变量来进行分面，因为这样更容易看到每个组中的模式：

```
sim3 <- sim3 %>%
  gather_residuals(mod1, mod2)

ggplot(sim3, aes(x1, resid, color = x2)) +
  geom_point() +
  facet_grid(model ~ x2)
```



mod2 的残差中几乎看不到明显的模式。mod1 的残差则表明这个模型在 b 分类中明显漏掉了某种模式，而在 c 和 d 分类中，虽然不明显，但还是存在某种模式的。你应该很想知道，是否有精确的方法可以确定 mod1 还是 mod2 更好。确实有这种方法，但需要强大的数学背景，而且我们现在还不用关心这个问题。现在我们应该关心的是定性评估模型能否捕获我们感兴趣的模式的方法。

17.4.3 交互项（两个连续变量）

以下模型和上一节中的基本相同，只是包括了两个连续变量。前几个步骤几乎和前面的示例完全一样：

```
mod1 <- lm(y ~ x1 + x2, data = sim4)
mod2 <- lm(y ~ x1 * x2, data = sim4)
grid <- sim4 %>%
  data_grid(
    x1 = seq_range(x1, 5),
    x2 = seq_range(x2, 5)
  ) %>%
  gather_predictions(mod1, mod2)
grid
#> # A tibble: 50 × 4
#>   model    x1    x2   pred
#>   <chr> <dbl> <dbl>  <dbl>
#> 1  mod1  -1.0  -1.0  0.996
#> 2  mod1  -1.0  -0.5 -0.395
#> 3  mod1  -1.0   0.0 -1.786
#> 4  mod1  -1.0   0.5 -3.177
#> 5  mod1  -1.0   1.0 -4.569
#> 6  mod1  -0.5  -1.0  1.907
#> # ... with 44 more rows
```

注意，我们在 data_grid() 函数中使用了 seq_range() 函数。这一次我们不使用 x 变量的所有唯一值，而是使用 x 变量最小值和最大值之间间隔相等的 5 个值来生成网格。虽然这种技术不是特别重要，但通常还是有用的。seq_range() 还有其他 3 个有用的参数。

- pretty = TRUE 会生成一个“漂亮的”序列，也就是说，让我们看起来比较舒服的序列。如果想要生成输出表格的话，这个参数是很有用的：

```
seq_range(c(0.0123, 0.923423), n = 5)
#> [1] 0.0123 0.2401 0.4679 0.6956 0.9234
seq_range(c(0.0123, 0.923423), n = 5, pretty = TRUE)
#> [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

- trim = 0.1 会截断 10% 的尾部值。如果变量具有长尾分布，而你希望尽量生成中心附近的值，那么就可以使用这个参数：

```
x1 <- rcauchy(100)
seq_range(x1, n = 5)
#> [1] -115.9   -83.5   -51.2   -18.8    13.5
seq_range(x1, n = 5, trim = 0.10)
#> [1] -13.84   -8.71   -3.58    1.55    6.68
seq_range(x1, n = 5, trim = 0.25)
```

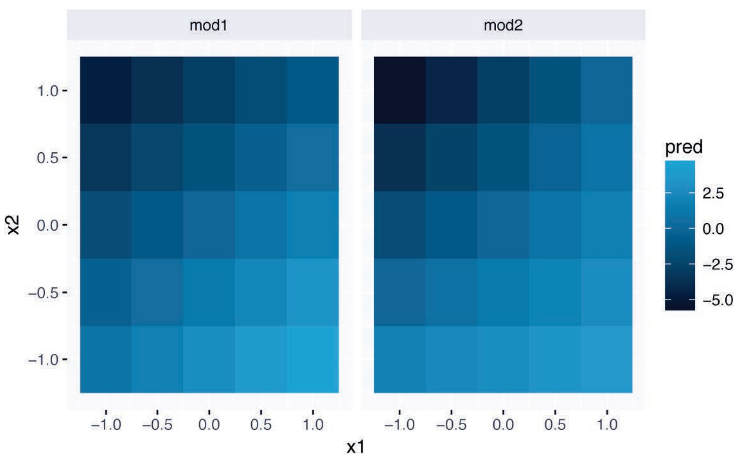
```
#> [1] -2.1735 -1.0594  0.0547  1.1687  2.2828
seq_range(x1, n = 5, trim = 0.50)
#> [1] -0.725 -0.268  0.189  0.647  1.104
```

- expand = 0.1 从某种程度上来说是 trim() 的反函数，它可以将取值范围扩大 10%：

```
x2 <- c(0, 1)
seq_range(x2, n = 5)
#> [1] 0.00 0.25 0.50 0.75 1.00
seq_range(x2, n = 5, expand = 0.10)
#> [1] -0.050  0.225  0.500  0.775  1.050
seq_range(x2, n = 5, expand = 0.25)
#> [1] -0.125  0.188  0.500  0.812  1.125
seq_range(x2, n = 5, expand = 0.50)
#> [1] -0.250  0.125  0.500  0.875  1.250
```

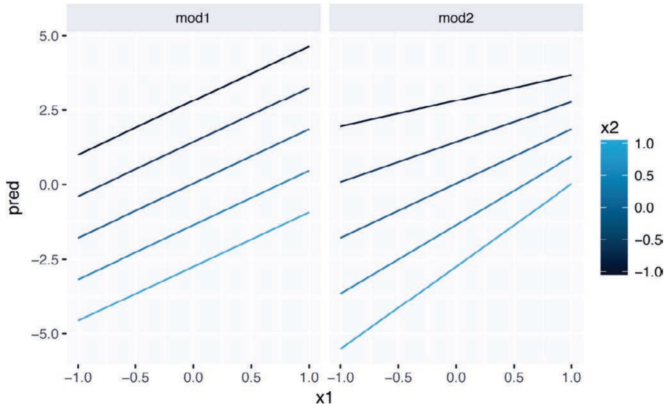
接下来我们试着对模型进行可视化。因为有两个连续型预测变量，所以我们可以将模型想象为一个三维表面。可以使用 geom_tile() 函数将其显示出来：

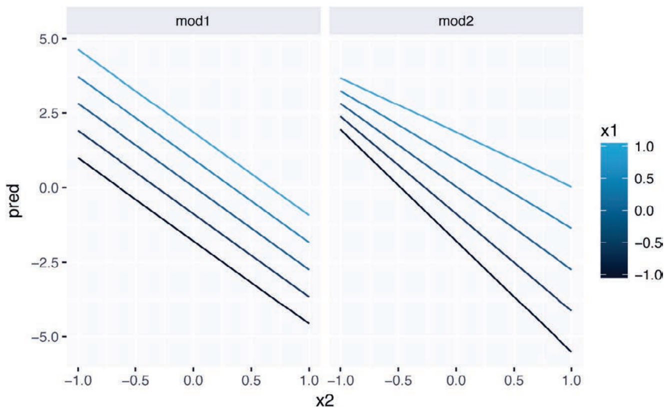
```
ggplot(grid, aes(x1, x2)) +
  geom_tile(aes(fill = pred)) +
  facet_wrap(~ model)
```



观察这两张图，我们看不出这两个模型有什么明显区别。但这是一种错觉，因为我们的眼睛和大脑不擅长精确分辨颜色的深浅。我们不能从上面查看这个表面，而应该分别从 x1 和 x2 的角度来查看，并表示出多个切面：

```
ggplot(grid, aes(x1, pred, color = x2, group = x2)) +
  geom_line() +
  facet_wrap(~ model)
ggplot(grid, aes(x2, pred, color = x1, group = x1)) +
  geom_line() +
  facet_wrap(~ model)
```





这说明两个连续型变量的交互项的作用方式与一个分类变量和一个连续变量的交互项基本相同。交互项说明了两个变量是相互影响的，如果要预测 y 值，那么必须同时考虑 `x1` 的值和 `x2` 的值。

由上可知，即使只有两个连续型变量，要想得到良好的可视化结果也是非常困难的。但这也是合乎情理的，对于 3 个或更多变量同时进行的交互作用，我们更不能指望可以轻松理解了。但再次让人聊以自慰的是，我们现在只是使用模型进行数据探索，以后还可以逐步完善这个模型。现在的模型不一定要很完美，只要能帮助我们更好地认识数据即可。

我们用了一点时间来检查残差，看看能否证明 `mod2` 的效果比 `mod1` 更好。结果是 `mod2` 的效果确实好一点，但真的只是一点。在后面的练习中，我们还会继续研究这个问题。

17.4.4 变量转换

还可以在模型公式中进行变量转换。例如，`log(y) ~ sqrt(x1) + x2` 可以转换为 `log(y) = a_1 + a_2 * sqrt(x1) + a_3 * x2`。如果想要使用 +、*、^ 或 - 进行变量转换，那么就应该使用 `I()` 对其进行包装，以便 R 在处理时不将它当作模型定义的一部分。例如，`y ~ x + I(x ^ 2)` 会转换为 `y = a_1 + a_2 * x + a_3 * x^2`。如果忘记使用 `I()`，将公式写成 `y ~ x ^ 2 + x`，那么 R 就会计算 `y ~ x * x + x`。`x * x` 表示 x 和自己的交互项，实际上就是

x。R 会自动丢弃冗余变量，因此 x + x 会变成 x，这样 y ~ x ^ 2 + x 定义的方程就是 y = a_1 + a_2 * x，这可不是你想要的！

再次强调，如果搞不清模型在做什么，可以使用 model_matrix() 函数查看 lm() 到底在拟合哪个方程：

```
df <- tribble(
  ~y, ~x,
   1,  1,
   2,  2,
   3,  3
)
model_matrix(df, y ~ x^2 + x)
#> # A tibble: 3 × 2
#>   `(Intercept)`     x
#>           <dbl> <dbl>
#> 1             1     1
#> 2             1     2
#> 3             1     3
model_matrix(df, y ~ I(x^2) + x)
#> # A tibble: 3 × 3
#>   `(Intercept)` `I(x^2)`     x
#>           <dbl>    <dbl> <dbl>
#> 1             1        1     1
#> 2             1        4     2
#> 3             1        9     3
```

变量转换非常有用，因为你可以使用它们来近似表示非线性函数。如果学过微积分，那么你就应该知道泰勒定理，它表明任何平滑函数都可以近似为无限个多项式之和。这说明通过拟合 y = a_1 + a_2 * x + a_3 * x^2 + a_4 * x^3 这样的方程，我们可以使用线性函数任意逼近一个平滑函数。因为手动输入这个方程太无聊了，所以 R 提供了一个辅助函数 poly()：

```
model_matrix(df, y ~ poly(x, 2))
#> # A tibble: 3 × 3
#>   `(Intercept)` `poly(x, 2)1` `poly(x, 2)2`
#>           <dbl>         <dbl>         <dbl>
#> 1             1     -7.07e-01         0.408
#> 2             1     -7.85e-17        -0.816
#> 3             1      7.07e-01         0.408
```

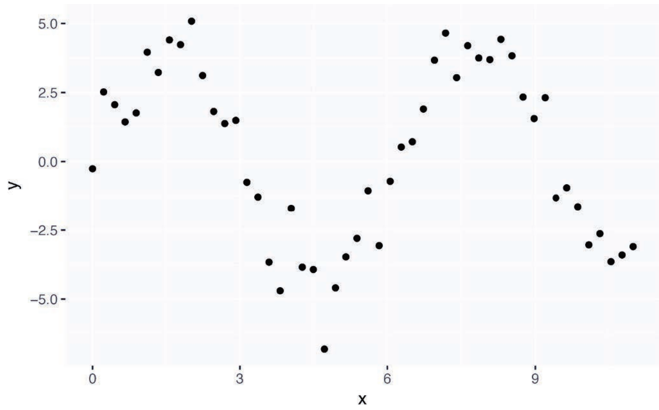
但使用 poly() 函数时有一个很大的问题：多项式的值会超出数据范围，很容易接近正无穷或负无穷。更安全的一种方式是使用自然样条法 splines::ns()：

```
library(splines)
model_matrix(df, y ~ ns(x, 2))
#> # A tibble: 3 × 3
#>   `(Intercept)` `ns(x, 2)1` `ns(x, 2)2`
#>           <dbl>       <dbl>       <dbl>
#> 1             1       0.000       0.000
#> 2             1       0.566      -0.211
#> 3             1       0.344       0.771
```

我们看一下近似非线性函数时的情况：

```r
sim5 <- tibble(
  x = seq(0, 3.5 * pi, length = 50),
  y = 4 * sin(x) + rnorm(length(x))
)

ggplot(sim5, aes(x, y)) +
  geom_point()
```

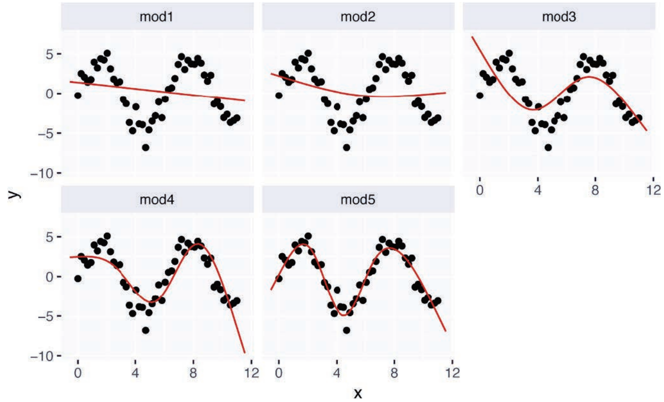


我们使用这份数据来拟合 5 个模型：

```r
mod1 <- lm(y ~ ns(x, 1), data = sim5)
mod2 <- lm(y ~ ns(x, 2), data = sim5)
mod3 <- lm(y ~ ns(x, 3), data = sim5)
mod4 <- lm(y ~ ns(x, 4), data = sim5)
mod5 <- lm(y ~ ns(x, 5), data = sim5)

grid <- sim5 %>%
  data_grid(x = seq_range(x, n = 50, expand = 0.1)) %>%
  gather_predictions(mod1, mod2, mod3, mod4, mod5, .pred = "y")

ggplot(sim5, aes(x, y)) +
  geom_point() +
  geom_line(data = grid, color = "red") +
  facet_wrap(~ model)
```



注意，当使用模型在数据范围外进行推断时，效果明显非常差。这是使用多项式近似函数的一个缺点。但这是所有模型都具有的一个实际问题：当对未知数据进行外推时，模型无法确保结果的真实性。你必须依靠相关的科学理论。

17.4.5 练习

(1) 如果使用没有截距的模型对 sim2 数据集再次进行分析，会是什么情况？模型方程会发生什么变化？预测值呢？

(2) 使用 model_matrix() 研究一下我们用 sim3 和 sim4 拟合模型时生成的方程。为什么 * 是交互项的一种非常好的简单表示？

(3) 使用前文中的基本规则，将以下两个模型中的公式转换为方程。（提示：先将分类变量转换为 0~1 变量。）

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

(4) 对于 sim4 数据集，mod1 和 mod2 哪一个更好？我们认为 mod2 在消除模式方面做得稍好一些，但好得很有限。你能否用一张图来支持该结论？

17.5 缺失值

对于变量间的关系，缺失值显然不能传达出任何信息，因此建模函数会丢弃包含缺失值的所有行。默认情况下，R 会不声不响地丢弃这种数据，但是 options(na.action = na.warn)（我们在准备工作中运行过这行代码）可以确保我们收到一条警告信息：

```
df <- tribble(
  ~x, ~y,
  1, 2.2,
  2, NA,
  3, 3.5,
  4, 8.3,
  NA, 10
)
mod <- lm(y ~ x, data = df)
#> Warning: Dropping 2 rows with missing values
```

要想阻止错误信息，可以设置 na.action = na.exclude：

```
mod <- lm(y ~ x, data = df, na.action = na.exclude)
```

通过 nobs() 函数，你可以知道模型实际使用了多少个观测：

```
nobs(mod)
#> [1] 3
```

17.6 其他模型族

本章的重点完全在于线性模型，其假设变量间的关系是 `y = a_1 * x1 + a_2 * x2 + … + a_n * xn` 的形式。线性模型还假设残差服从正态分布，我们之前没有讲过这一点。很多模型族以各种有趣的方式扩展了线性模型，其中包括以下各种模型。

- **广义线性模型**，如 `stats::glm()` 函数。线性模型假设响应变量是连续的，并且误差服从正态分布。广义线性模型将线性模型扩展为可以包括非连续型的响应变量（如二值数据或计数）。它们基于似然这一统计思想，通过定义距离的度量来工作。
- **广义可加模型**，如 `mgcv::gam()` 函数。该模型可以扩展广义线性模型，使其包含任意的平滑函数。这意味着，你可以写出 `y ~ s(x)` 这样的公式，它可以转化为 `y = f(x)` 这样的方程，并使用 `gam()` 函数估计出方程的形式（由于某些平滑性条件的限制，这个问题还是比较容易解决的）。
- **带有惩罚项的线性模型**，如 `glmnet::glmnet()` 函数。该模型向距离添加一个惩罚项，以惩罚复杂的模型（使用参数向量和原点间的距离来定义）。在扩展到来自同一总体的新数据集时，这种方法生成的模型更容易取得良好的效果。
- **健壮线性模型**，如 `MASS:rlm()`。该模型对过远的距离进行调整，以降低远距离的权重。这样一来，模型对异常值就不会太敏感，但代价是当没有异常值时，效果不是特别好。
- **树模型**，如 `rpart::rpart()`。该模型使用与线性模型完全不同的方法来解决问题。树模型拟合一个分段常数模型，将数据逐渐划分为越来越小的多个部分。树模型本身的效率不是特别高，但如果使用**随机森林**（random forest，如 `randomForest::randomForest()` 函数）或**梯度提升机**（gradient boosting machine，如 `xgboost::xgboost()` 函数）这样的模型将它们聚合起来使用时，其功能是非常强大的。

从编程的角度来看，这些模型的工作原理非常相似。一旦掌握了线性模型，你就会发现很容易掌握其他模型族的运行机制。要想成为一名熟练的建模专家，既要掌握通用原则，又要精通大量建模技术。现在你已经学习了一些通用工具和一种有用的模型，接下来就可以通过其他资源学习更多种类的模型了。

第 18 章

模型构建

18.1 简介

我们在上一章中学习了线性模型的工作原理，还学习了一些基本工具来理解模型如何帮助我们解释数据。上一章主要使用模拟数据集来帮助我们学习模型是如何工作的。本章则关注真实数据，介绍如何循序渐进地建立模型以帮助我们理解数据。

我们将利用这样一个共识：模型可以将数据分成模式与残差这两个部分。我们会先利用数据可视化找出模式，然后通过模型更加具体而精确地提取出模式。之后会重复这一过程，只是将原来的响应变量替换为模型的残差。我们的目标是将数据与头脑中的隐式知识转换为量化模型中的显式知识。这样更容易将知识应用于新的领域，也更容易为他人所使用。

对于特别大和特别复杂的数据集，这将是一项繁重的工作。当然还有其他方法，即重点在于模型预测能力的机器学习方法。这些方法往往会产生黑盒效应：模型的预测效果非常好，但你无法解释。这些方法确实很合理，但很难将你的实际知识应用于模型。因此，从长期来看，如果基本情况发生了变化，我们就很难确定模型是否还会奏效。对于多数实际模型，我们希望你能将这种方法和一些更经典的自动化方法结合起来使用。

适可而止是很困难的。你应该知道模型什么时候是恰到好处的，什么时候是画蛇添足、过犹不及的。我们非常欣赏 reddit 用户 Broseidon241 说的以下这段话。

> 很久之前的艺术课上，老师告诉我："艺术家应该知道何时完成作品。如果不能做得更好，那就结束它。如果不喜欢它，那就从头再来。否则，就去做别的事情吧。"在后来的生活中，我还听到了这句话："坏裁缝会犯很多错误，好裁缝会努力纠正错误，而优秀的裁缝则从来不怕将有问题的衣服扔掉，重新开始。"
>
> ——Broseidon241

准备工作

我们使用的工具与上一章中的相同，但要加入几个实际数据集：ggplot2 包中的 `diamonds` 和 nycflights13 包中的 `flights`。我们还需要 lubridate 包，以处理 `flights` 数据集中的日期和时间数据。

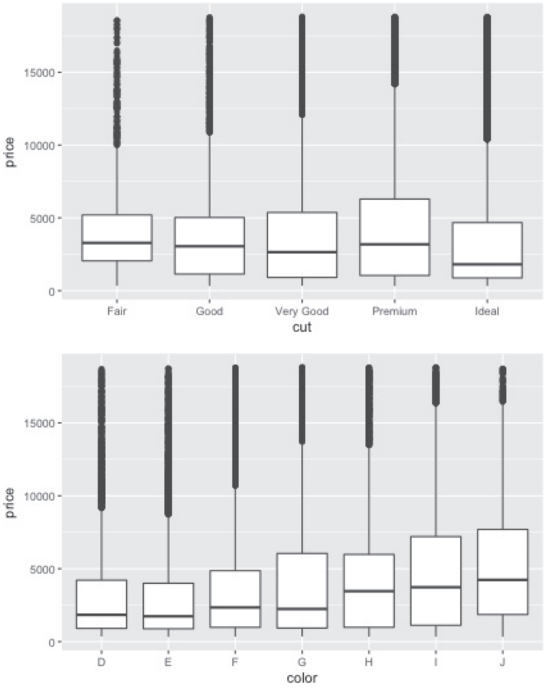
```
library(tidyverse)
library(modelr)
options(na.action = na.warn)

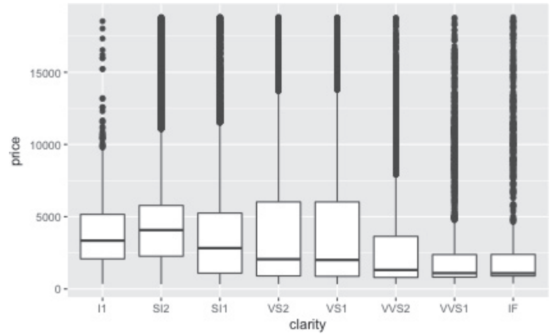
library(nycflights13)
library(lubridate)
```

18.2　为什么质量差的钻石更贵

在前面的章节中，我们已经发现了钻石质量与价格间这种令人惊讶的关系：质量差的钻石（切工差、颜色差、纯净度低）具有更高的价格：

```
ggplot(diamonds, aes(cut, price)) + geom_boxplot()
ggplot(diamonds, aes(color, price)) + geom_boxplot()
ggplot(diamonds, aes(clarity, price)) + geom_boxplot()
```



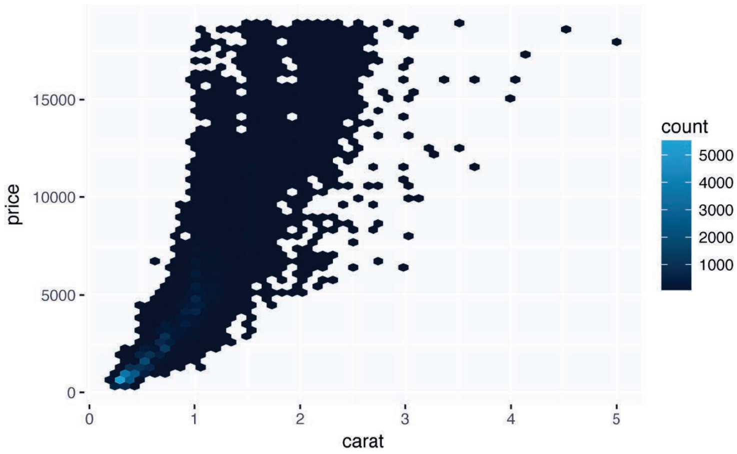


注意，最差的钻石颜色是 J（微黄），最差的纯净度是 I1（肉眼可见内含物）。

18.2.1 价格与重量

质量差的钻石似乎价格更高，造成这一现象的原因是一个重要的混淆变量：钻石的重量（carat）。重量是确定钻石价格的单一因素中最重要的一个，而质量差的钻石往往更重一些：

```
ggplot(diamonds, aes(carat, price)) +
  geom_hex(bins = 50)
```



通过拟合一个模型来分离出 carat 变量的作用，我们可以更容易看到钻石的其他特性对 price 的影响。但是，我们需要先对钻石数据集进行一些调整，以便其更容易处理。

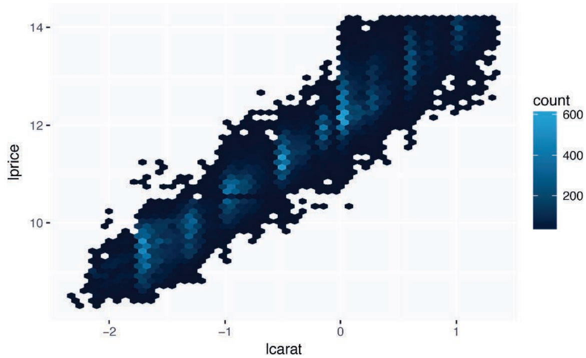
(1) 重点关注小于 2.5 克拉的那些钻石（全部数据的 99.7%）。

(2) 对重量和价格变量进行对数转换。

```
diamonds2 <- diamonds %>%
  filter(carat <= 2.5) %>%
  mutate(lprice = log2(price), lcarat = log2(carat))
```

这两个调整可以让我们更轻松地看到 carat 和 price 之间的关系：

```
ggplot(diamonds2, aes(lcarat, lprice)) +
  geom_hex(bins = 50)
```



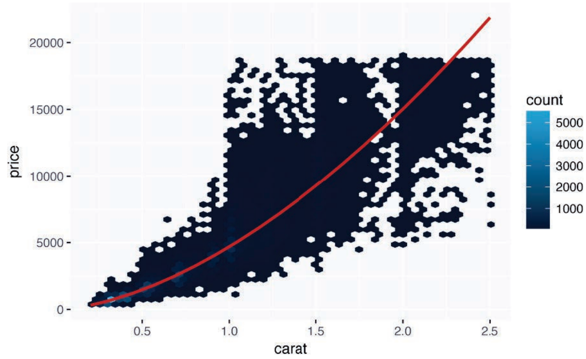
对数转换在这个示例中非常有用，因为它可以让模式变为线性的，而线性模式是最容易处理的。现在我们进行下一步，从数据中去除这种强烈的线性模式。我们通过拟合一个模型让这种模式成为显式的：

```
mod_diamond <- lm(lprice ~ lcarat, data = diamonds2)
```

接着我们检查模型，看看它能够反映出数据中的哪些信息。注意，因为我们对预测值进行了反向变换，还原了对数转换，所以可以将预测值覆盖在原始数据上：

```
grid <- diamonds2 %>%
  data_grid(carat = seq_range(carat, 20)) %>%
  mutate(lcarat = log2(carat)) %>%
  add_predictions(mod_diamond, "lprice") %>%
  mutate(price = 2 ^ lprice)

ggplot(diamonds2, aes(carat, price)) +
  geom_hex(bins = 50) +
  geom_line(data = grid, color = "red", size = 1)
```

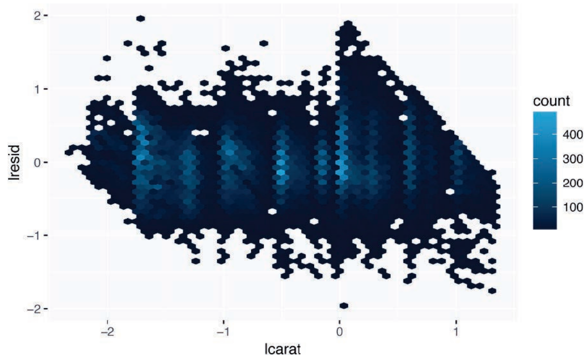


这张图可以告诉我们关于这份数据的一些有趣信息。如果我们相信这个模型，那么大钻石要比预料中便宜得多。这可能是因为数据集中没有价格超过 $19 000 的钻石。

现在我们可以检查一下残差，它可以用来验证我们是否成功移除了强烈的线性模式：

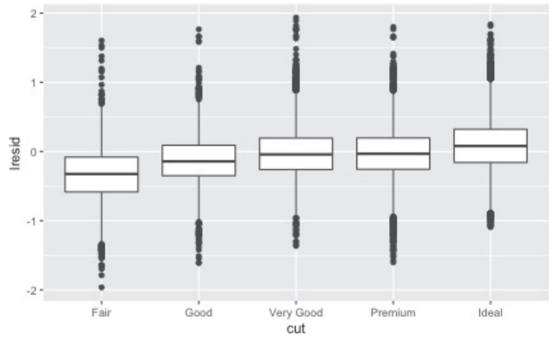
```
diamonds2 <- diamonds2 %>%
  add_residuals(mod_diamond, "lresid")

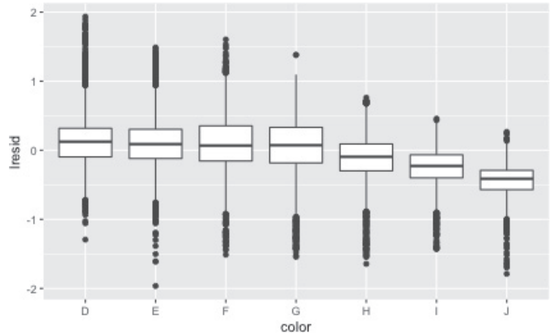
ggplot(diamonds2, aes(lcarat, lresid)) +
  geom_hex(bins = 50)
```

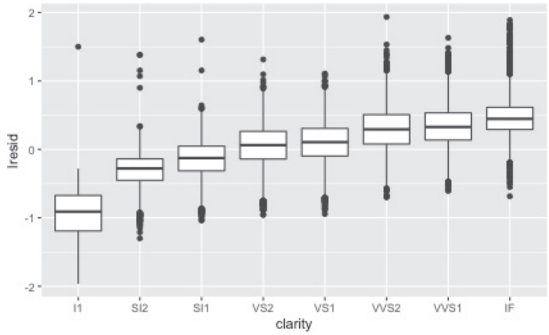


重要的是，我们现在可以使用残差代替 price 来重新绘图了：

```
ggplot(diamonds2, aes(cut, lresid)) + geom_boxplot()
ggplot(diamonds2, aes(color, lresid)) + geom_boxplot()
ggplot(diamonds2, aes(clarity, lresid)) + geom_boxplot()
```







现在我们可以看到期望中的关系了：当钻石的质量下降时，其相应价格也随之下降。为了解释 *y* 轴，我们需要思考一下残差的意义及其使用的标度。残差为 –1 表示 lprice 比仅使用重量进行估计的预测值少一个单位。2^{-1} 就是 1/2，因此值为 –1 的点的价格为预计价格的一半，残差为 1 时，价格则是预计价格的 2 倍。

18.2.2 一个更复杂的模型

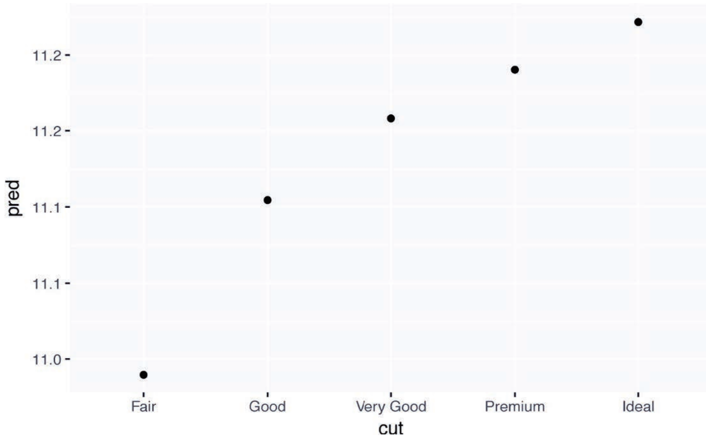
如果愿意的话，我们可以继续构建模型，用模型明确表示观察到的效果。例如，我们可以在模型中包括 color、cut 和 clarity 变量，以将这 3 个分类变量的效果明确表示出来：

```
mod_diamond2 <- lm(
  lprice ~ lcarat + color + cut + clarity,
  data = diamonds2
)
```

现在模型中包括了 4 个预测变量，因此更加难以进行可视化。好在这些变量还是彼此独立的，这意味着我们可以在 4 张图中分别绘制出它们。为了让这个过程更简单一些，我们在 data_grid() 函数中使用 .model 参数：

```
grid <- diamonds2 %>%
  data_grid(cut, .model = mod_diamond2) %>%
  add_predictions(mod_diamond2)
grid
#> # A tibble: 5 × 5
#>         cut lcarat color clarity  pred
#>       <ord>  <dbl> <chr>   <chr> <dbl>
#> 1      Fair -0.515     G     SI1  11.0
#> 2      Good -0.515     G     SI1  11.1
#> 3 Very Good -0.515     G     SI1  11.2
#> 4   Premium -0.515     G     SI1  11.2
#> 5     Ideal -0.515     G     SI1  11.2

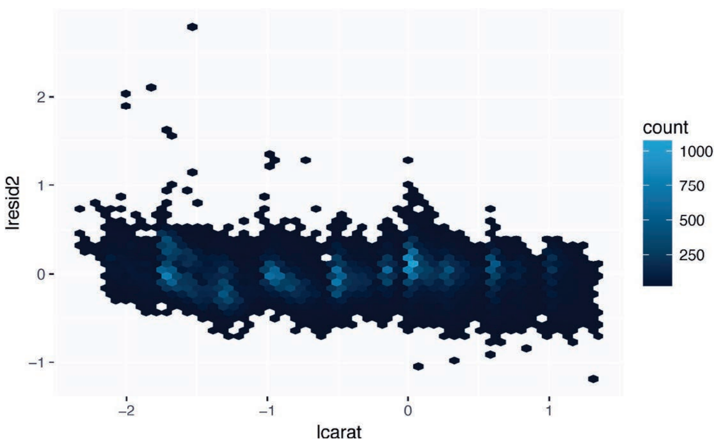
ggplot(grid, aes(cut, pred)) +
  geom_point()
```



如果模型需要你还没有明确提供的变量，data_grid() 函数会自动使用“典型”值来填充它们。对于连续变量，模型使用中位数；对于分类变量，模型使用最常见的值（或多个值，如果有同样数量的多个值的话）：

```
diamonds2 <- diamonds2 %>%
  add_residuals(mod_diamond2, "lresid2")

ggplot(diamonds2, aes(lcarat, lresid2)) +
  geom_hex(bins = 50)
```



这张图说明一些钻石有非常大的残差。记住，残差为 2 表示钻石的价格是预计价格的 4 倍。通常还应该检查一下异常值：

```
diamonds2 %>%
  filter(abs(lresid2) > 1) %>%
  add_predictions(mod_diamond2) %>%
  mutate(pred = round(2 ^ pred)) %>%
  select(price, pred, carat:table, x:z) %>%
  arrange(price)
#> # A tibble: 16 × 11
```

```
#>   price  pred carat     cut color clarity depth table     x
#>   <int> <dbl> <dbl>   <ord> <ord>   <ord> <dbl> <dbl> <dbl>
#> 1  1013   264  0.25    Fair     F     SI2  54.4    64  4.30
#> 2  1186   284  0.25 Premium     G     SI2  59.0    60  5.33
#> 3  1186   284  0.25 Premium     G     SI2  58.8    60  5.33
#> 4  1262  2644  1.03    Fair     E      I1  78.2    54  5.72
#> 5  1415   639  0.35    Fair     G     VS2  65.9    54  5.57
#> 6  1415   639  0.35    Fair     G     VS2  65.9    54  5.57
#> # ... with 10 more rows, and 2 more variables: y <dbl>,
#> #  z <dbl>
```

这个结果没什么大价值，但或许我们可以花点时间思考一下出现异常值是因为模型有问题，还是数据中有错误。如果是数据中的错误，那么我们就有机会买到那些错误定了低价的钻石。

18.2.3　练习

(1) `lcarat` 和 `lprice` 的关系图中有一些垂直的亮条。它们表示什么？

(2) 如果 `log(price) = a_0 + a_1 * log(carat)`，这能反映出 `price` 和 `carat` 间的何种关系？

(3) 提取残差特别大和特别小的钻石数据，这些钻石有什么特异之处？它们是特别好、特别差，还是定价错误？

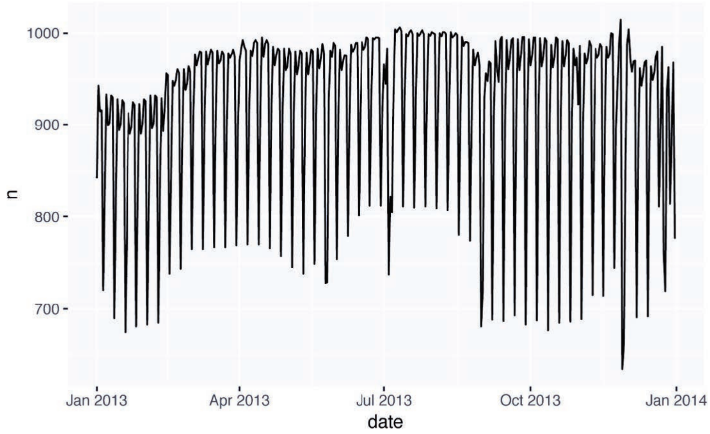
(4) 最终模型 `mod_diamonds2` 是预测钻石价格的优秀模型吗？如果想要买钻石，你会信任它给出的价格吗？

18.3　哪些因素影响了每日航班数量

我们使用同样的流程来处理乍看上去更为简单的一个数据集：每天从纽约市出发的航班数量。这是一个相当小的数据集，只有 365 行和 2 列，而且我们也不准备充分实现最终模型。但正如你将看到的，实现模型的一系列步骤会帮助我们更加透彻地理解数据。首先，我们需要计算每天出发的航班数量，并使用 ggplot2 进行可视化：

```
daily <- flights %>%
  mutate(date = make_date(year, month, day)) %>%
  group_by(date) %>%
  summarize(n = n())
daily
#> # A tibble: 365 × 2
#>         date     n
#>       <date> <int>
#> 1 2013-01-01   842
#> 2 2013-01-02   943
#> 3 2013-01-03   914
#> 4 2013-01-04   915
#> 5 2013-01-05   720
#> 6 2013-01-06   832
#> # ... with 359 more rows
```

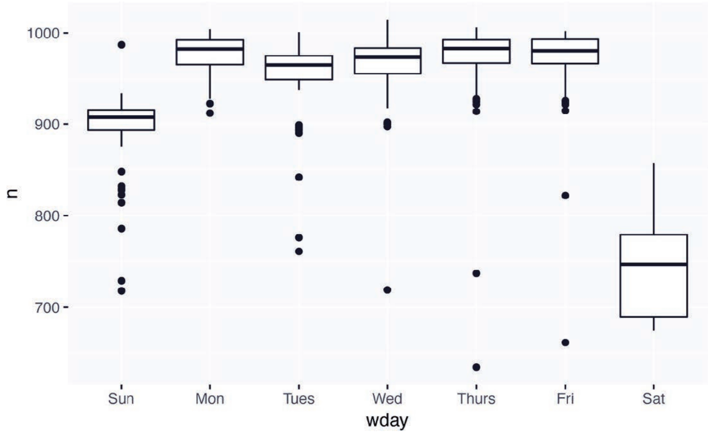
```
ggplot(daily, aes(date, n)) +
  geom_line()
```



18.3.1 一周中的每一天

理解长期趋势是非常困难的，因为数据中存在着强烈的周内效应，它严重影响了数据中的微妙模式。我们先检查一下航班数量在一周中的每一天的分布：

```
daily <- daily %>%
  mutate(wday = wday(date, label = TRUE))
ggplot(daily, aes(wday, n)) +
  geom_boxplot()
```



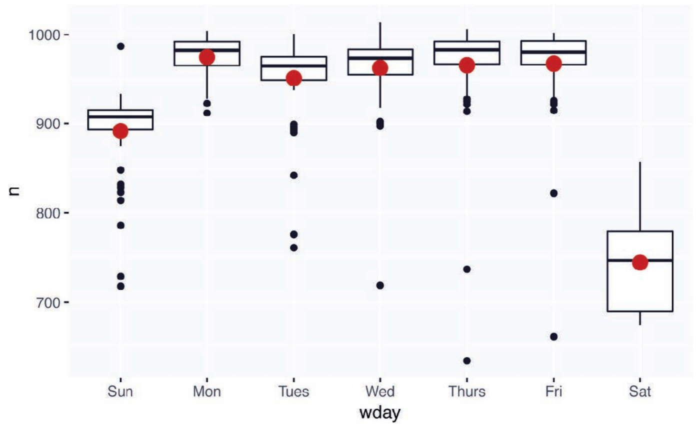
周末的航班数量更少，因为多数行程都是公务出差。这种效应在星期六体现的更加明显：有时为了星期一的会议，人们或许会在星期日出发，但很少有人会在星期六出发，因为这天我们更愿意留在家里陪伴家人。

去除这种强烈模式的一种方法是使用模型。首先，我们拟合这个模型，并将预测值覆盖在原始数据上：

```
mod <- lm(n ~ wday, data = daily)

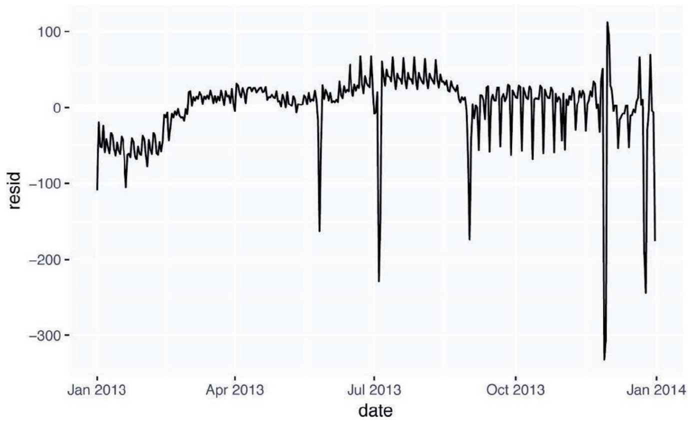
grid <- daily %>%
  data_grid(wday) %>%
  add_predictions(mod, "n")

ggplot(daily, aes(wday, n)) +
  geom_boxplot() +
  geom_point(data = grid, color = "red", size = 4)
```



然后计算残差，并对其进行可视化表示：

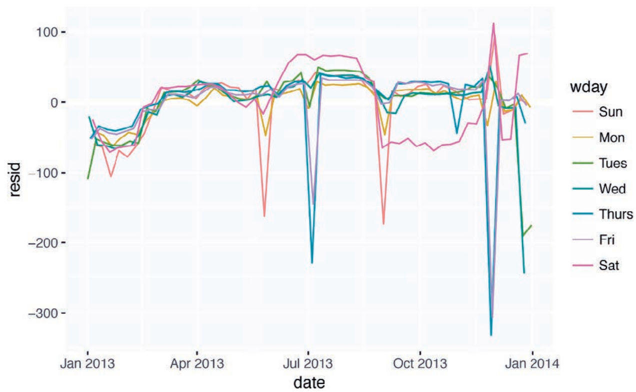
```
daily <- daily %>%
  add_residuals(mod)
daily %>%
  ggplot(aes(date, resid)) +
  geom_ref_line(h = 0) +
  geom_line()
```



注意 y 轴的变化：现在它表示的是每天的航班数量与给定一周中某一天预期航班数量间的偏差。这张图是非常有用的，因为它去除了大部分周内效应，我们可以从剩余的信息中看出一些微妙的模式。

- 我们的模型似乎从 6 月份开始失效了：你可以看到，此时数据中仍然存在一种强烈的、有规律的模式，这是我们没有捕获到的。我们再绘制一张图，将一周中的每一天都用一条折线表示出来，这样可以看得更清楚一些：

```
ggplot(daily, aes(date, resid, color = wday)) +
  geom_ref_line(h = 0) +
  geom_line()
```



我们的模型没有精确地预测出星期六的航班数量：夏季的实际航班数量比我们预计的要多，秋季则比我们预计的要少。下一节将介绍如何更好地捕获这种模式。

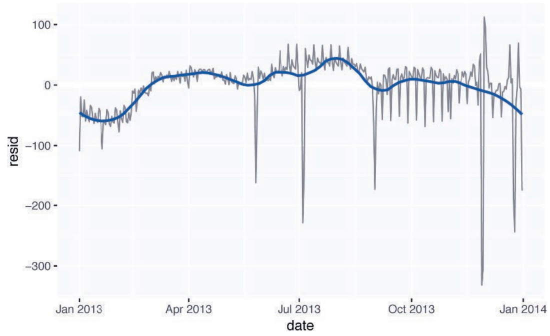
- 有几天的航班数量远远少于预期：

```
daily %>%
  filter(resid < -100)
#> # A tibble: 11 × 4
#>         date     n  wday resid
#>       <date> <int> <ord> <dbl>
#> 1 2013-01-01   842  Tues  -109
#> 2 2013-01-20   786   Sun  -105
#> 3 2013-05-26   729   Sun  -162
#> 4 2013-07-04   737 Thurs  -229
#> 5 2013-07-05   822   Fri  -145
#> 6 2013-09-01   718   Sun  -173
#> # ... with 5 more rows
```

如果非常熟悉美国的公共假期，那么你就会发现这些日期中包括新年、7 月 4 日、感恩节和圣诞节。还有一些其他日期没有对应的公共假期，我们会在后面的练习中继续讨论这些日期。

- 从整年来看，似乎有某种更平滑的长期趋势。我们可以使用 geom_smooth() 函数来高亮显示这种趋势：

```
daily %>%
  ggplot(aes(date, resid)) +
  geom_ref_line(h = 0) +
  geom_line(color = "grey50") +
  geom_smooth(se = FALSE, span = 0.20)
#> `geom_smooth()` using method = 'loess'
```

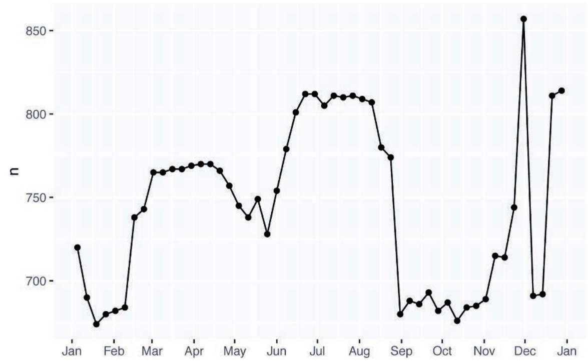


可以看出，1 月（和 12 月）的航班比较少，而夏季（5~9 月）的航班比较多。不能对这种模式进行更多量化处理，因为我们只有一年的数据。但是，我们可以使用领域知识尽情地想象各种可能的解释。

18.3.2 季节性星期六效应

我们先解决未能精确预测星期六航班数量的问题。先回到原始数据，关注星期六的航班状况，这不失为一种良好的做法：

```
daily %>%
  filter(wday == "Sat") %>%
  ggplot(aes(date, n)) +
    geom_point() +
    geom_line() +
    scale_x_date(
      NULL,
      date_breaks = "1 month",
      date_labels = "%b"
    )
```



（我们同时使用了点和折线，这样可以更清楚地表示哪些是数据，哪些是插值。）

我们怀疑这种模式是由暑假造成的：很多人夏季出去度假，而假期中的人们是不介意在星期六出行的。根据这张图，我们可以猜想暑假是从 6 月初到 8 月末，这个时间与美国的学校假期非常吻合：2013 年的暑假是从 6 月 26 日至 9 月 9 日。

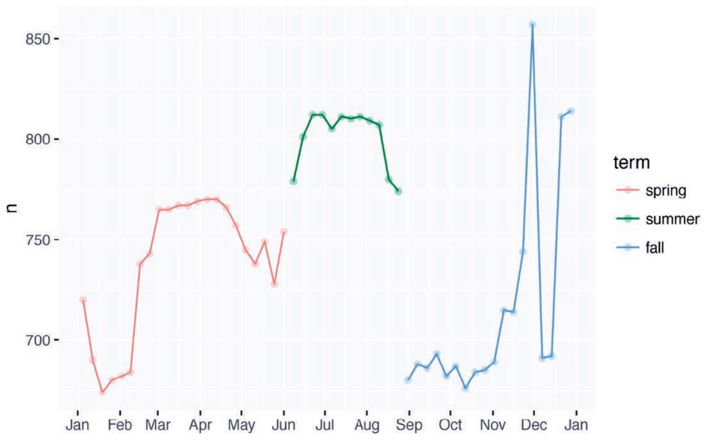
为什么春季的星期六航班比秋季多？我们询问了一些美国朋友，他们认为之所以较少在秋季安排全家度假，是因为还有很长的感恩节和圣诞节假期。虽然没有数据证明这种说法的真实性，但这似乎是个还算合理的解释。

我们创建一个“学期”变量来粗略地表示学校的 3 个学期，然后使用图形检查一下这样做的效果：

```
term <- function(date) {
  cut(date,
    breaks = ymd(20130101, 20130605, 20130825, 20140101),
    labels = c("spring", "summer", "fall")
  )
}

daily <- daily %>%
  mutate(term = term(date))

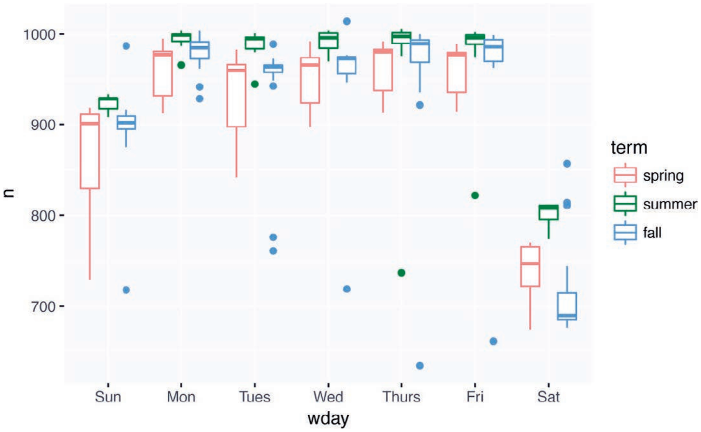
daily %>%
  filter(wday == "Sat") %>%
  ggplot(aes(date, n, color = term)) +
  geom_point(alpha = 1/3) +
  geom_line() +
  scale_x_date(
    NULL,
    date_breaks = "1 month",
    date_labels = "%b"
  )
```



（为了让图形中的间隔更美观，我们手动调整了日期。使用可视化图形可以帮助我们理解函数的作用，这确实是一种强大又通用的技术。）

还应该查看这个新变量是如何影响一周中其他各天的：

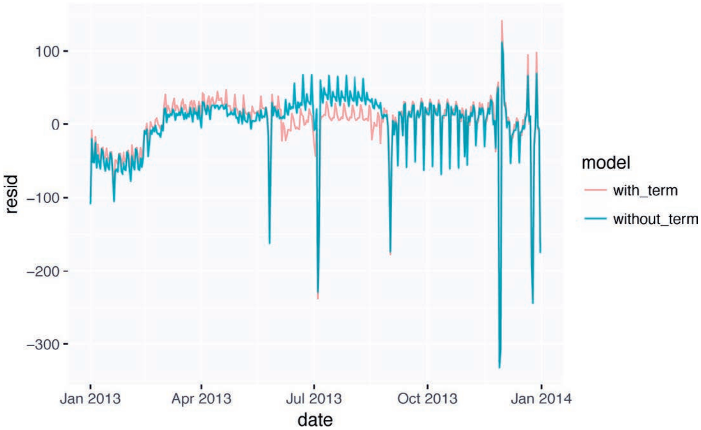
```
daily %>%
  ggplot(aes(wday, n, color = term)) +
    geom_boxplot()
```



看上去不同学期间的差别还是非常大的，因此应该拟合去除了每学期周内效应的模型。这样可以改进模型，但效果只是差强人意：

```
mod1 <- lm(n ~ wday, data = daily)
mod2 <- lm(n ~ wday * term, data = daily)

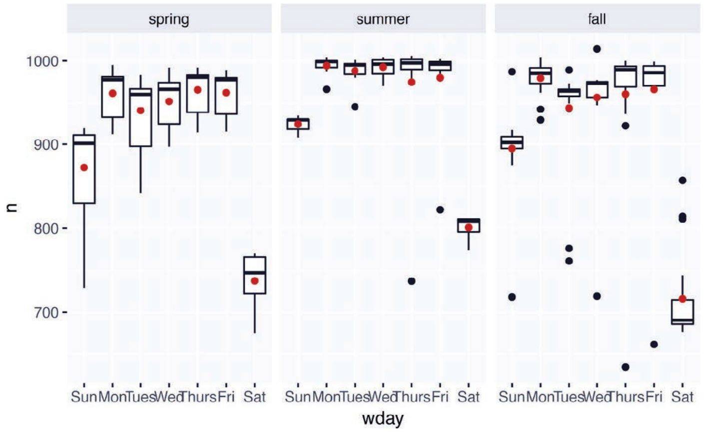
daily %>%
  gather_residuals(without_term = mod1, with_term = mod2) %>%
  ggplot(aes(date, resid, color = model)) +
    geom_line(alpha = 0.75)
```



将模型预测值覆盖在原始数据上，我们就可以看出问题所在：

```
grid <- daily %>%
  data_grid(wday, term) %>%
  add_predictions(mod2, "n")

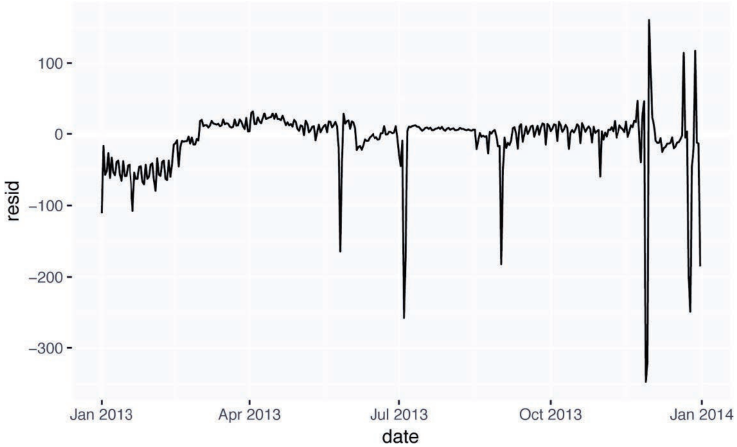
ggplot(daily, aes(wday, n)) +
  geom_boxplot() +
  geom_point(data = grid, color = "red") +
  facet_wrap(~ term)
```



模型寻找的是**平均效应**，但我们的数据中有大量数值很大的离群点，因此平均趋势与典型值之间的差别比较大。如果想要改善这个问题，可以使用对离群点健壮的模型：MASS::rlm()。这个函数可以大大减轻离群点对模型估计的影响，并给出很好地消除了周内效应的一个模型：

```
mod3 <- MASS::rlm(n ~ wday * term, data = daily)

daily %>%
  add_residuals(mod3, "resid") %>%
  ggplot(aes(date, resid)) +
  geom_hline(yintercept = 0, size = 2, color = "white") +
  geom_line()
```



现在更容易看出长期趋势，以及正负离群点了。

18.3.3　计算出的变量

如果正在试验多个模型和多种可视化方法，那么你可以将创建变量的所有代码打包放在一个函数中。这是一种非常好的做法，可以防止由于意外而在不同的地方执行了不同的转

换。例如，我们可以使用以下代码：

```
compute_vars <- function(data) {
  data %>%
    mutate(
      term = term(date),
      wday = wday(date, label = TRUE)
    )
}
```

另一种方法是直接在模型公式中进行转换：

```
wday2 <- function(x) wday(x, label = TRUE)
mod3 <- lm(n ~ wday2(date) * term(date), data = daily)
```

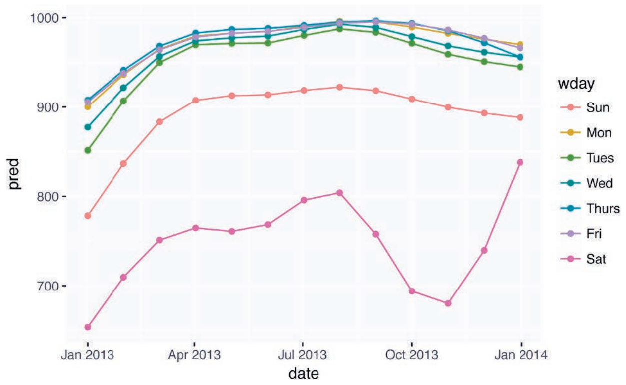
每种方法都有其合理性。如果想对工作进行检查，或者想对工作结果进行可视化表示，那么你就应该明确表示变量转换。谨慎使用返回多个列的那些转换（比如样条法）。如果正在处理多个不同的数据集，那么将转换放在模型公式中可以使得工作更容易一些，因为这时的模型是自成一体的。

18.3.4　年度时间：另一种方法

我们在前面使用了领域知识（美国学校学期对旅行的影响）改进模型。改进模型的另一种方法是赋予数据更多的发言权。可以使用一种更灵活的模型来捕获我们所关注的模式。简单的线性趋势已经无法满足我们的需要了，我们试着使用自然样条法拟合一个年度平滑曲线模型：

```
library(splines)
mod <- MASS::rlm(n ~ wday * ns(date, 5), data = daily)

daily %>%
  data_grid(wday, date = seq_range(date, n = 13)) %>%
  add_predictions(mod) %>%
  ggplot(aes(date, pred, color = wday)) +
    geom_line() +
    geom_point()
```



我们可以在星期六航班数量中看到一种强烈的模式。这很令人欣慰，因为也可以在原始数据中看到这种模式。通过不同方法得到同样的信号，是一种很好的迹象。

18.3.5 练习

(1) 通过你的搜索技能进行头脑风暴，解释为什么 1 月 20 日、5 月 26 日和 9 月 1 日的航班数量比预计的要少。（提示：它们的解释相同。）如何能够将这些日期推广到另一个年度？

(2) 以下具有最大正残差的 3 个日期代表什么？如何能够将这些日期推广到另一个年度？

```
daily %>%
  top_n(3, resid)
#> # A tibble: 3 × 5
#>         date     n  wday resid   term
#>       <date> <int> <ord> <dbl> <fctr>
#> 1 2013-11-30   857   Sat 112.4   fall
#> 2 2013-12-01   987   Sun  95.5   fall
#> 3 2013-12-28   814   Sat  69.4   fall
```

(3) 创建一个新变量将 `wday` 变量划分为几个学期，但仅对星期六执行这个操作。也就是说，可以保留 `Thurs`、`Fri` 等值，但要将星期六拆分为 `Sat-summer`、`Sat-spring` 和 `Sat-fall`。使用这种方法拟合出的模型与使用 `wday` 和 `term` 的所有组合拟合出的模型有什么不同？

(4) 创建一个新的 `wday` 变量，将一周 7 天、（星期六的）学期因素和公共假期组合起来，对于这个新变量拟合出的模型，其残差是怎样的？

(5) 如果在考虑月份的情况下拟合周内效应（即 `n ~ wday * month`），那么会是什么情况？为什么这种模型用处不大？

(6) 你觉得 `n ~ wday + ns(date, 5)` 这个模型的效果会怎么样？基于对数据的理解，为什么你会觉得这个模型不会特别有效？

(7) 我们进行了这样一个假设：星期日出发的人们多数都是出差，需要在星期一到达某个地方。仔细研究一下这个假设，看看如何基于航班时间与距离来证明这个假设是错误的：如果这个假设正确，那么数据中就应该有更多在星期日晚上出发的远途航班。

(8) 在前面的图形中，星期日和星期六分别处于图形的两端，这样给人的感觉不太好。编写一个小函数，重新设置因子水平，以便一周从星期一开始。

18.4 学习更多模型知识

本章只是对建模进行了一点蜻蜓点水式的介绍，但我们确实希望你能从中获取一些简单而又通用的技能，以提高自己的数据分析水平。千里之行，始于足下！如你所见，即使是非常简单的模型，如果有能力使用不同方法来处理变量，那么也会得到差异非常大的结果。

与本书其他部分相比，关于建模内容的这部分更多地掺杂了个人观点。我们从与大多数人不同的角度来构建模型，而且本书留给建模的篇幅确实有些少。其实需要一本专门的著作来介绍建模，因此，我们强烈建议你至少阅读以下三本书中的一本。

- *Statistical Modeling: A Fresh Approach*，Danny Kaplan 著。这本书既对建模进行了简单明了的介绍，也可以帮助你建立直觉、掌握数学工具和 R 语言技能。这本书不是那种传统的“统计学入门”教材，而是提供了与数据科学相关的最新内容。
- *An Introduction to Statistical Learning*，Gareth James、Daniela Witten、Trevor Hastie 及 Robert Tibshirani 著（有免费的在线版）。这本书介绍了称为“统计学习”的一整套现代建模技术。如果想要加深对模型的数学理解，可以阅读 Trevor Hastie、Robert Tibshirani 和 Jerome Friedman 的经典著作 *Elements of Statistical Learning*（有免费的在线版）。
- *Applied Predictive Modeling*，Max Kuhn 及 Kjell Johnson 著。这本书是对 caret 包的重要补充，提供了多种实用工具，可以帮助你解决实际工作的预测性建模难题。

第 19 章

使用purrr和broom处理多个模型

19.1 简介

本章将介绍 3 种功能强大的方法来帮助我们游刃有余地处理大量模型。

- 使用多个简单模型来更好地理解复杂数据集。
- 使用列表列在数据框中保存任意数据结构。例如，可以通过这种方法让数据列中包含线性模型。
- 使用 David Robinson 开发的 broom 包将模型转换为整洁数据。这是一种非常强大的技术，可以处理大量模型，因为一旦有了整洁数据，我们在本书前面学到的所有技术就有用武之地了。

以下各节将详细介绍本章要使用的各种技术。

- 19.2 节将介绍关于列表列的数据结构知识，以及可以将列表放在数据框中的原因。
- 19.3 节将介绍创建列表列的 3 种主要方法。
- 19.4 节将介绍如何将列表列还原为常用的原子向量（或原子向量集合），以便更容易进行处理。
- 19.5 节将介绍由 broom 包提供的一整套工具，以及它们在其他类型数据结构上的使用方法。

本章属于进阶内容，如果本书是你的第一本 R 语言书，那么本章将是一个严峻挑战。它需要你对建模、数据结构和迭代都有深刻理解。因此，如果看不懂本章内容，也不要担心——先将它放在一边，如果几个月后想做做思维训练，就再来读读看。

准备工作

处理多个模型需要 tidyverse 中的多个包（分别用于数据探索、纷争和编程），以及帮助建模的 modelr 包。

```
library(modelr)
library(tidyverse)
```

19.2 列表列

本节将详细地研究列表列的数据结构。直到最近，我们才真正领会列表列的设计思想。列表列是隐式定义在数据框中的：数据框是由相同长度的向量组成的命名列表。一个列表就是一个向量，因此将列表作为数据框的一列是完全合理的。但是，在 R 基础包中创建列表列是非常困难的，而且 `data.frame()` 函数是将列表作为列的列表来处理的：

```
data.frame(x = list(1:3, 3:5))
#>   x.1.3 x.3.5
#> 1     1     3
#> 2     2     4
#> 3     3     5
```

要想 `data.frame()` 不这样处理，可以使用 `I()` 函数，但是输出结果却是难以理解的：

```
data.frame(
  x = I(list(1:3, 3:5)),
  y = c("1, 2", "3, 4, 5")
)
#>         x       y
#> 1 1, 2, 3    1, 2
#> 2 3, 4, 5 3, 4, 5
```

tibble 更懒惰一些（`tibble()` 不对输入进行修改），但更容易创建列表列，输出结果也更容易理解：

```
tibble(
  x = list(1:3, 3:5),
  y = c("1, 2", "3, 4, 5")
)
#> # A tibble: 2 × 2
#>           x       y
#>      <list>   <chr>
#> 1 <int [3]>    1, 2
#> 2 <int [3]> 3, 4, 5
```

使用 `tribble()` 函数则更容易，因为它可以自动识别出你想要的列表：

```
tribble(
   ~x, ~y,
  1:3, "1, 2",
  3:5, "3, 4, 5"
)
#> # A tibble: 2 × 2
```

```
#>          x       y
#>      <list>   <chr>
#> 1 <int [3]>    1, 2
#> 2 <int [3]> 3, 4, 5
```

列表列的最大用处是作为一种中间数据结构。要想直接处理列表列是比较困难的，因为大多数 R 函数只能处理原子向量或数据框。但列表列可以将相关项目统一保存在一个数据框中，仅这一个优点就值得我们花些精力来学习它。

一般来说，要想有效地使用列表列，需要 3 个步骤。

(1) 使用 19.3 节中介绍的 3 种方法之一来创建列表列。这 3 种方法是：`nest()`、`summarize()` + `list()` 以及 `mutate()` + 映射函数。

(2) 通过使用 `map()`、`map2()` 或 `pmap()` 函数转换现有列表列，创建一个中间列表列。例如，在前面的案例研究中，我们通过转换数据框的列表列创建了一个模型列表列。

(3) 使用 19.4 节中介绍的方法，将列表列简化还原成数据框或原子向量。

19.3　创建列表列

通常来说，我们不会使用 `tibble()` 函数创建列表列，而是使用以下 3 种方法之一，根据普通列进行创建。

(1) 使用 `tidyr::nest()` 函数将分组数据框转换为嵌套数据框，嵌套数据框中会包含数据框列表列。

(2) 使用 `mutate()` 函数以及能够返回列表的向量化函数。

(3) 使用 `summarize()` 函数以及能够返回多个结果的摘要函数。

另外，我们还可以使用 `tibble::enframe()` 函数根据命名列表来创建列表列。

通常来说，在创建列表列时，应该确保这些列是同质的，即列中的每个元素都包含同样类型的内容。R 不对这种要求进行检查，但如果想要对列表列使用 purrr 函数或者要求类型一致的函数时，你就会发现这种要求是顺理成章的。

19.3.1　使用嵌套

可以使用 `nest()` 函数创建嵌套数据框，即带有数据框列表列的数据框。在嵌套数据框中，每行都是一个元观测：除列表列外的列给出了定义观测的变量，数据框中的列表列则给出了组成元观测的具体观测。

`nest()` 函数有两种使用方式。当用于分组数据框时，`nest()` 函数会保留用于分组的列，而将其他所有数据归并到列表列中。

还可以在未分组数据框上使用 `nest()`，此时需要指定嵌套哪些列。

19.3.2 使用向量化函数

有些常用函数接受一个原子向量并返回一个列表。例如，我们在第 10 章中学习了 stringr::str_split() 函数，它接受一个字符向量，并返回字符向量的一个列表。如果在 mutate() 函数中使用这个函数，那么就会得到一个列表列：

```
df <- tribble(
  ~x1,
  "a,b,c",
  "d,e,f,g"
)

df %>%
  mutate(x2 = stringr::str_split(x1, ","))
#> # A tibble: 2 × 2
#>        x1        x2
#>     <chr>    <list>
#> 1   a,b,c <chr [3]>
#> 2 d,e,f,g <chr [4]>
```

unnest() 函数知道如何处理这些向量列表：

```
df %>%
  mutate(x2 = stringr::str_split(x1, ",")) %>%
  unnest()
#> # A tibble: 7 × 2
#>        x1    x2
#>     <chr> <chr>
#> 1   a,b,c     a
#> 2   a,b,c     b
#> 3   a,b,c     c
#> 4 d,e,f,g     d
#> 5 d,e,f,g     e
#> 6 d,e,f,g     f
#> # ... with 1 more rows
```

（如果会大量使用这种嵌套还原功能，请一定研究一下 tidyr::separate_rows() 函数，其包装了这种常见的、将一列拆分成多行的功能。）

使用向量化函数创建列表列的另一个示例是使用 purrr 包中的 map()、map2() 和 pmap() 函数。例如，我们再看一下 16.7 节中的“调用不同函数”中的最后一个示例，并重写代码以应用 mutate() 函数：

```
sim <- tribble(
  ~f,      ~params,
  "runif", list(min = -1, max = -1),
  "rnorm", list(sd = 5),
  "rpois", list(lambda = 10)
)

sim %>%
  mutate(sims = invoke_map(f, params, n = 10))
#> # A tibble: 3 × 3
#>       f     params       sims
```

```
#>   <chr>     <list>     <list>
#> 1 runif <list [2]> <dbl [10]>
#> 2 rnorm <list [1]> <dbl [10]>
#> 3 rpois <list [1]> <int [10]>
```

注意，从技术角度来说，`sim` 并不是同质的，因为其中既有双精度型向量，也有整型向量。但是，因为整型向量和双精度型向量都是数值型向量，所以不会有什么大问题。

19.3.3　使用多值摘要

`summarize()` 函数的一个局限性是，只能使用返回单一值的摘要函数。这意味着我们不能使用像 `quantile()` 这样的函数，因为它会返回任意长度的向量：

```
mtcars %>%
  group_by(cyl) %>%
  summarize(q = quantile(mpg))
#> Error in eval(expr, envir, enclos): expecting a single value
```

然而，你可以将结果包装在一个列表中！这是符合 `summarize()` 函数的约定的，因为这样每个摘要结果就是一个长度为 1 的列表（向量）了：

```
mtcars %>%
  group_by(cyl) %>%
  summarize(q = list(quantile(mpg)))
#> # A tibble: 3 × 2
#>     cyl         q
#>   <dbl>    <list>
#> 1     4 <dbl [5]>
#> 2     6 <dbl [5]>
#> 3     8 <dbl [5]>
```

要想让 `unnest()` 函数的结果更可用，我们还需要表示出概率：

```
probs <- c(0.01, 0.25, 0.5, 0.75, 0.99)
mtcars %>%
  group_by(cyl) %>%
  summarize(p = list(probs), q = list(quantile(mpg, probs))) %>%
  unnest()
#> # A tibble: 15 × 3
#>     cyl     p     q
#>   <dbl> <dbl> <dbl>
#> 1     4  0.01  21.4
#> 2     4  0.25  22.8
#> 3     4  0.50  26.0
#> 4     4  0.75  30.4
#> 5     4  0.99  33.8
#> 6     6  0.01  17.8
#> # ... with 9 more rows
```

19.3.4　使用命名列表

带有列表列的数据框可以解决一种常见问题：如何同时对列表的元素及元素内容进行迭

代？相对于将所有元素内容塞进一个对象，更容易的一种方法是创建一个数据框：一列包含元素名称，另一列包含元素中的列表内容。创建这种数据框的一种简单方法是使用 `tibble::enframe()` 函数：

```
x <- list(
  a = 1:5,
  b = 3:4,
  c = 5:6
)
df <- enframe(x)
df
#> # A tibble: 3 × 2
#>    name      value
#>   <chr>     <list>
#> 1     a <int [5]>
#> 2     b <int [2]>
#> 3     c <int [2]>
```

这种结构的优点是可以用非常简单的方式进行扩展，如果有元数据字符向量，那么名称是很有用的，但如果有其他类型的数据或多个向量，那么名称就没有什么作用了。

如果想要同时对名称和值进行迭代，那么可以使用 `map2()` 函数：

```
df %>%
  mutate(
    smry = map2_chr(
      name,
      value,
      ~ stringr::str_c(.x, ": ", .y[1])
    )
  )
#> # A tibble: 3 × 3
#>    name      value  smry
#>   <chr>     <list> <chr>
#> 1     a <int [5]>  a: 1
#> 2     b <int [2]>  b: 3
#> 3     c <int [2]>  c: 5
```

19.3.5 练习

(1) 列举你能想到的接受一个原子向量并返回一个列表的所有函数。

(2) 进行头脑风暴，列举返回多个值的所有摘要函数，如 `quantile()`。

(3) 以下数据框丢失了什么信息？如何使 `quantile()` 函数返回那些丢失的数据？为什么那些数据在这里不是很重要？

```
mtcars %>%
  group_by(cyl) %>%
  summarize(q = list(quantile(mpg))) %>%
  unnest()
#> # A tibble: 15 × 2
#>     cyl     q
#>   <dbl> <dbl>
```

```
#> 1     4  21.4
#> 2     4  22.8
#> 3     4  26.0
#> 4     4  30.4
#> 5     4  33.9
#> 6     6  17.8
#> # ... with 9 more rows
```

(4) 以下代码的作用是什么？为什么它或许会有用？

```
mtcars %>%
  group_by(cyl) %>%
  summarize_each(funs(list))
```

19.4　简化列表列

为了应用在本书中学到的数据处理和可视化技术，我们需要将列表列简化还原为一个普通列（即一个原子向量）或一组普通列。在将列表列还原为简单结构时，根据想要将列表列的一个元素转换为单个值还是多个值，再决定使用哪种技术。

- 如果想要得到单个值，就使用 mutate() 以及 map_lgl()、map_int()、map_dbl() 和 map_chr() 函数来创建一个原子向量。
- 如果想要得到多个值，就使用 unnest() 函数将列表列还原为普通列，这样可以按需要将行多次重复。

以下各节将更加详细地介绍这两种技术。

19.4.1　列表转换为向量

如果可以将列表列缩减为一个原子向量，那么这个原子向量就可以作为一个普通列。例如，你可以总是使用类型和长度来描述一个对象，因此对于所有列表列，以下代码都可以运行：

```
df <- tribble(
  ~x,
  letters[1:5],
  1:3,
  runif(5)
)
df %>% mutate(
  type = map_chr(x, typeof),
  length = map_int(x, length)
)
#> # A tibble: 3 × 3
#>           x      type length
#>      <list>     <chr>  <int>
#> 1 <chr [5]> character      5
#> 2 <int [3]>   integer      3
#> 3 <dbl [5]>    double      5
```

使用默认的表格打印方法同样可以得到这些基本信息，但现在你就可以使用它们进行筛选操作了。如果你有一个异构列表，并且想要筛选掉其中不需要的部分，那么就可以使用这

种方法。

别忘了，我们还可以使用 `map_*()` 快捷方式，例如，你可以使用 `map_chr(x, "apple")` 从 x 的每个元素中提取变量 apple 中的内容。这种方法可以提取嵌套列表中的一部分，并使结果成为普通列。如果元素中有缺失值，还可以使用 `.null` 参数提供一个返回值（而不是返回 NULL）：

```
df <- tribble(
  ~x,
  list(a = 1, b = 2),
  list(a = 2, c = 4)
)
df %>% mutate(
  a = map_dbl(x, "a"),
  b = map_dbl(x, "b", .null = NA_real_)
)
#> # A tibble: 2 × 3
#>            x     a     b
#>       <list> <dbl> <dbl>
#> 1 <list [2]>     1     2
#> 2 <list [2]>     2    NA
```

19.4.2 嵌套还原

`unnest()` 函数对列表列中的每个元素都重复一次普通列。例如，在以下这个非常简单的示例中，我们将第一行重复了 4 次（因为 y 中的第一个元素的长度是 4），而第二行只重复了 1 次：

```
tibble(x = 1:2, y = list(1:4, 1)) %>% unnest(y)
#> # A tibble: 5 × 2
#>       x     y
#>   <int> <dbl>
#> 1     1     1
#> 2     1     2
#> 3     1     3
#> 4     1     4
#> 5     2     1
```

这意味着你不能同时还原包含不同数量元素的两个列表列：

```
# 以下代码可以运行，因为y和z每行中的元素数量都相同
df1 <- tribble(
  ~x, ~y,           ~z,
   1, c("a", "b"), 1:2,
   2, "c",           3
)
df1
#> # A tibble: 2 × 3
#>       x          y          z
#>   <dbl>     <list>     <list>
#> 1     1 <chr [2]> <int [2]>
#> 2     2 <chr [1]> <dbl [1]>
df1 %>% unnest(y, z)
```

```
#> # A tibble: 3 × 3
#>       x     y     z
#>   <dbl> <chr> <dbl>
#> 1     1     a     1
#> 2     1     b     2
#> 3     2     c     3

# 以下代码不能运行，因为y和z每行中的元素数量不同
df2 <- tribble(
  ~x, ~y,           ~z,
   1, "a",         1:2,
   2, c("b", "c"),   3
)
df2
#> # A tibble: 2 × 3
#>       x          y          z
#>   <dbl>     <list>     <list>
#> 1     1 <chr [1]> <int [2]>
#> 2     2 <chr [2]> <dbl [1]>
df2 %>% unnest(y, z)
#> Error: All nested columns must have
#> the same number of elements.
```

数据框列表列的还原也遵循同样的原则。只要每行中数据框的行数都相同，那么你就可以同时还原多个列表列。

19.4.3　练习

(1) 为什么可以使用 `lengths()` 函数根据列表列创建原子向量？

(2) 列举数据框中最常用的向量类型。列表和数据框有什么不同？

19.5　使用broom生成整洁数据

broom 包提供了 3 种常用工具，用于将模型转换为整洁数据框。

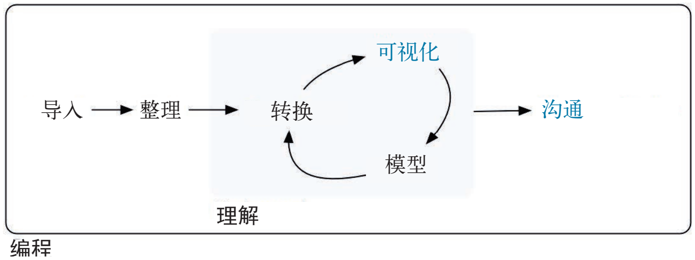
- `broom::glance(model)` 为每个模型返回一行数据，其中每一列都是模型的一个摘要统计量：要么是模型质量的度量方式，要么是模型复杂度，又或者是二者的组合。
- `broom::tidy(model)` 为模型的每个系数返回一行数据，其中每一列都是系数的估计值或变异指标。
- `broom::augment(model, data)` 返回 `data` 中的每一行，但会添加一些额外信息，如残差以及其他一些有影响的统计量。

broom 的适用范围很广，支持大量常用建模包所生成的模型。可以搜索并查看一下 broom 现在支持的模型列表。

第五部分

沟通

到目前为止，我们已经学习了如何将数据导入 R、如何将数据整理为方便分析的形式，以及如何通过转换、可视化与建模来理解数据。但除非能向他人解释结果，否则无论你的数据分析做得多么精彩，都没什么用处。因此，你需要与他人**沟通**工作成果。



沟通是这一部分 4 章内容的主题。

- 第 20 章将介绍 R Markdown，这是一种集成文本、代码和结果的工具。R Markdown 提供了笔记本模式供不同的分析者进行沟通与交流，还提供了报告模式供分析者和决策者进行沟通与交流。由于 R Markdown 文件的强大功能，你甚至可以使用同一个文档来同时实现上述两种目的。
- 第 21 章将介绍如何将探索性图形转换为解释性图形。解释性图形可以帮助人们尽可能快速又轻松地理解你的分析工作。
- 第 22 章将介绍使用 R Markdown 可生成的多种输出，其中包括仪表盘、网站和书籍。
- 第 23 章是本书的最后一章，将介绍如何制作"分析式笔记本"，以及如何系统地记录工作中的成功和失败，以便总结经验和教训。

遗憾的是，以上各章重点关注的是沟通的技术机制，而不是人与人之间想法的实际沟通。但是，我们将在每章末尾介绍一些关于实际沟通的优秀著作。

第20章

R Markdown

20.1 简介

R Markdown 为数据科学提供了一种统一的写作框架，可以集成代码、输出结果和文本注释。R Markdown 文档是完全可重用的，并支持多种输出形式，包括 PDF、Word、幻灯片等。

R Markdown 文件的应用方式有 3 种。

- 与决策者沟通交流，这部分人关注的是最终结论，而不是分析背后的代码。
- 与其他数据科学家（包括未来的你）协作，这部分人关心的内容既包括最终结论，也包括得到结论的过程（也就是代码）。
- 作为开展数据科学工作的一种环境，它是一种现代化的实验室记录工具，不但可以记录你的实际工作，还可以记录你的思考过程。

R Markdown 集成了一些 R 包和外部工具。这意味着基本上不能再使用 ? 获取帮助了。因此，当学习本章或未来使用 R Markdown 进行工作时，你最好将以下资料放在手边。

- R Markdown 速查表：在 RStudio IDE 中就可以找到，路径为 Help → Cheatsheets → R Markdown Cheat Sheet。
- R Markdown 用户指南：在 RStudio IDE 中就可以找到，路径为 Help → Cheatsheets → R Markdown Reference Guide。

准备工作

我们需要使用 rmarkdown 包，但不用显式地安装或加载这个包，因为 RStudio 可以在我们需要时自动完成这些工作。

20.2 R Markdown基础

以下就是一个 R Markdown 文件，一个扩展名为 .Rmd 的纯文本文件：

```
---
title: "Diamond sizes"
date: 2016-08-25
output: html_document
---

```{r setup, include = FALSE}
library(ggplot2)
library(dplyr)

smaller <- diamonds %>%
  filter(carat <= 2.5)
```

We have data about `r nrow(diamonds)` diamonds. Only
`r nrow(diamonds) - nrow(smaller)` are larger than
2.5 carats. The distribution of the remainder is shown
below:

```{r, echo = FALSE}
smaller %>%
  ggplot(aes(carat)) +
  geom_freqpoly(binwidth = 0.01)
```
```

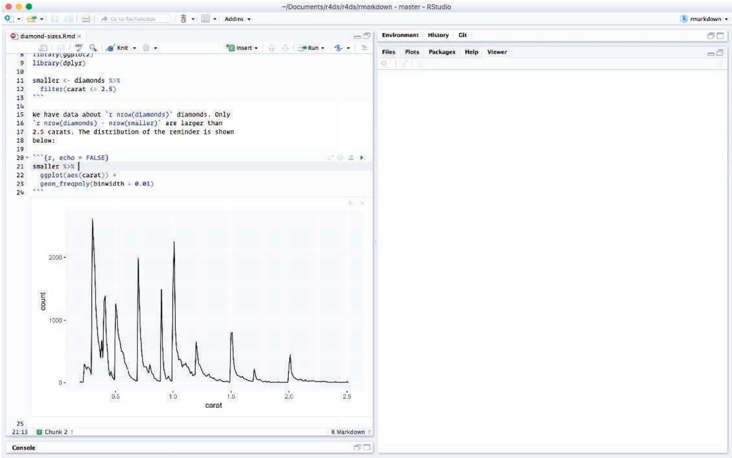
这个文件包含了 3 种重要的内容类型。

(1) 两个 --- 之间的（可选）YAML **文件头**。

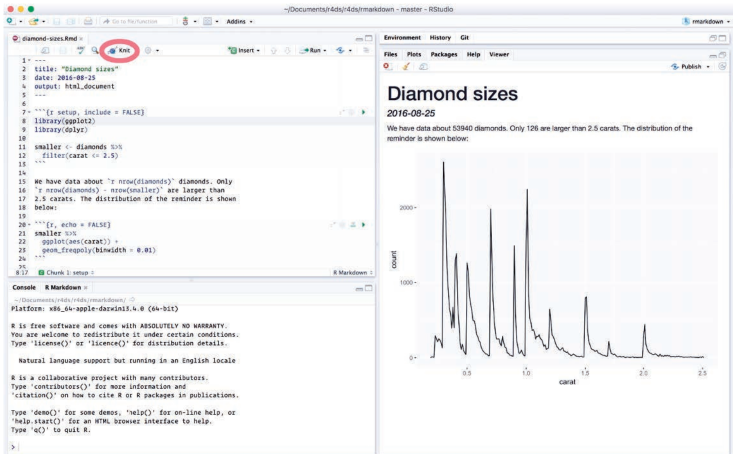
(2) 两个 ``` 之间的 R **代码段**。

(3) 一些具有简单格式的文本，比如前面用 # 表示的文本标题，以及两个 _ 之间的斜体。

打开一个 .Rmd 文件后，就会进入笔记本界面，其中代码和代码结果可以交互显示。通过点击 Run 图标（位于代码段上方，类似播放按钮）或按组合键 Ctrl+Shift+Enter，就可以运行每个代码段。RStudio 可以运行代码，并将运行结果显示在代码下面。



如果想要生成包含所有文本、代码和输出的完整报告，可以点击 Knit 或按组合键 Ctrl+Shift+K。还可以使用代码 `rmarkdown::render("1-example.Rmd")` 在程序中完成这个操作。报告会显示在查看器窗格中，并生成可以与他人分享的一个独立的 HTML 文件。



在生成文档时，R Markdown 先将 .Rmd 文件发送给 knitr，knitr 会执行所有代码段，并创建一个新的 Markdown 文件（.md），其中包含所有代码和输出。然后 knitr 生成的 Markdown 文件再由 pandoc 进行处理，并生成最终文件。这种两阶段工作流的优点是可以创建多种输出格式，第 22 章将介绍这些格式。



如果想要创建自己的 .Rmd 文件，可以在菜单栏选择 File → New File → R Markdown…… RStudio 会启动一个向导程序，对文件预先填充一些有用的内容，并对如何使用 R Markdown 的核心功能作出提示。

以下几节将详细介绍 R Markdown 文件的 3 个主要组成部分：Markdown 文本、代码段和 YAML 文件头。

练习

(1) 使用 File → New File → R Notebook 创建一个新的 Notebook 文件。阅读操作指示，并实际运行代码段。确认你能够修改代码、重新运行代码，并检查修改后的运行结果。

(2) 使用 File → New File → R Markdown 创建一个新的 R Markdown 文件。使用正确的按钮生成这个文件。使用正确的组合键生成这个文件。确认你能够修改输入，并获得更新后的输出。

(3) 比较并对比你在前面创建的 R Notebook 文件和 R Markdown 文件。它们的输出有什么相同之处？有什么不同之处？它们的输入有什么相同之处？有什么不同之处？如果将一个文件的 YAML 文件头复制到另一个文件中，会发生什么情况？

(4) 对于 3 种内置格式 HTML、PDF 和 Word，分别创建一个新的 R Markdown 文件，并生成这 3 种文件。这 3 种文件的输出有什么不同？输入有什么不同？（为了生成 PDF 输出，需要安装 LaTex，RStudio 会在需要时提醒你。）

20.3 使用Markdown格式化文本

.Rmd 文件中的文本是用 Markdown 语言写成的，Markdown 是用于格式化纯文本文件的一种轻量级语法，其设计思想就是使得文本既容易书写又容易阅读。Markdown 学习起来非常容易，以下指南给出了使用 Pandoc Markdown 的方法，这是 R Markdown 可以理解的 Markdown 的一种扩展版本：

```
Text formatting
------------------------------------------------------------

*italic*  or _italic_
**bold**   __bold__
`code`
superscript^2^ and subscript~2~

Headings
------------------------------------------------------------

# 1st Level Header

## 2nd Level Header

### 3rd Level Header

Lists
------------------------------------------------------------

*   Bulleted list item 1

*   Item 2

    * Item 2a

    * Item 2b

1.  Numbered list item 1

1.  Item 2. The numbers are incremented automatically in
the output.

Links and images
------------------------------------------------------------

<http://example.com>
[linked phrase](http://example.com)

![optional caption text](path/to/img.png)
```

```
Tables
------------------------------------------------------------

First Header  | Second Header
------------- | -------------
Content Cell  | Content Cell
Content Cell  | Content Cell
```

学习 Markdown 语法的最好方法就是练习。可能需要几天的时间，但你很快就能熟悉这种语法，无须思考就可以熟练自如地使用。如果忘记了某种语法，可以使用 Help → Markdown Quick Reference 来获取帮助。

练习

(1) 通过创建个人简历来练习使用 R Markdown 文件。一级标题是你的姓名，并且至少要包括以下二级标题：教育背景和工作经历。每个二级标题下都应该包括一个无序列表，列出你的学位和工作信息。使用粗体来强调年份。

(2) 使用 R Markdown 快速参考来找出以下操作方法。

a. 添加一个脚注。
b. 添加一条水平分隔线。
c. 添加一个块级引用。

(3) 从 https://github.com/hadley/r4ds/tree/master/rmarkdown 复制 diamond-sizes.Rmd 文件，并粘贴到本地的一个 R Markdown 文件中。确认这个文件可以运行，然后在频率多边形图下面添加一些文本来描述这个图中最引人注目的特征。

20.4　代码段

要想在 R Markdown 文件中运行代码，你需要插入代码段。插入代码段的方法有 3 种。

(1) 使用组合键 Ctrl+Alt+I。

(2) 使用编辑器工具栏上的 Insert 按钮。

(3) 手工输入代码段标记符 `` ```{r} `` 和 `` ``` ``。

显然，我们建议你使用组合键，因为长远来看，这可以节省大量时间。

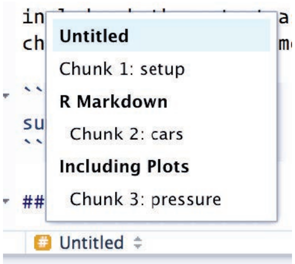
接下来，你可以使用迄今为止我们（希望你也是）最喜欢的一个组合键来运行代码：Ctrl+Enter。还可以使用一个新的组合键：Ctrl+Shift+Enter，这个组合键可以运行代码段内的全部代码。我们可以将代码段看作一个函数，它是相对独立的，而且专注于实现某个特定任务。

以下各节将介绍代码段的头部，它由 `` ```{r `` 开头，接着是一个可选的代码段名称，然后是由逗号分隔的代码段选项，再然后是 `}`。接下来就是具体的 R 代码。代码段的结束标志是 `` ``` ``。

20.4.1　代码段名称

我们可以赋予代码段一个可选的名称：```{r by-name}。这样做有以下 3 个优势。

- 使用脚本编辑器左下角的弹出式代码浏览菜单，更方便浏览特定的代码段。



- 可以使得代码段生成的图形具备有意义的名称，从而更容易在其他地方使用。21.7.2 节将更加深入地介绍这一点。
- 你可以建立缓存代码段的网络，以避免每次运行都重新调用昂贵的计算资源。以下内容将对此进行更多介绍。

`setup` 这个代码段名称具有特殊意义。当处于笔记本模式时，名称为 `setup` 的代码段会在任何其他代码运行前自动运行一次。

20.4.2　代码段选项

可以使用**选项**来定制代码段输出，选项是提供给代码段头部的参数。knitr 提供了差不多 60 种选项，你可以使用这些选项来定制自己的代码段。接下来将介绍几个最重要的代码段选项，你以后会经常用到这些选项。可以在 http://yihui.name/knitr/options/ 中找到完整的选项列表。

以下是最重要的一组选项，用来控制代码段是否可以执行，以及最终报告中包括哪些结果。

- `eval = FALSE` 禁止对代码进行求值。（很明显，如果代码不能运行的话，就不能生成什么结果。）在显示示例代码或不通过每行注释禁用大段代码时，这个选项是非常有用的。
- `include = FALSE` 可以运行代码，但不会在最终文档中显示代码和结果。如果不想让 `setup` 代码出现在报告中，就可以使用这个选项。
- `echo = FALSE` 禁止代码出现在最终报告中，但不会禁止结果。为不想看到 R 代码的人们编写报告时，就可以使用这个选项。
- `message = FALSE` 或 `warning = FALSE` 可以防止消息或警告出现在最终报告中。
- `results = 'hide'` 可以隐藏文本输出；`fig.show = 'hide'` 可以隐藏图形输出。
- `error = TRUE` 在代码出现错误时仍然可以生成最终报告。在报告的最终版中，我们很少需要包括出错信息，但在调试 .Rmd 文件时，出错信息是非常有用的。如果使用 R 进行教学活动，并特意想要包括出错信息的话，这个选项是非常有用的。如果使用默认设置 `error = FALSE`，那么即使只有一个错误，文档生成也会失败。

下表总结了每个选项对输出的具体控制。

选项	运行代码	显示代码	输出	图形	消息	警告
`eval = FALSE`	x		x	x	x	x
`include = FALSE`		x	x	x	x	x
`echo = FALSE`		x				
`results = "hide"`			x			
`fig.show = "hide"`				x		
`message = FALSE`					x	
`warning = FALSE`						x

20.4.3 表格

默认情况下，R Markdown 输出数据框和矩阵的格式与我们在控制台中看到的相同：

```
mtcars[1:5, 1:10]
#>                    mpg cyl disp  hp drat   wt qsec vs am gear
#> Mazda RX4         21.0   6  160 110 3.90 2.62 16.5  0  1    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.88 17.0  0  1    4
#> Datsun 710        22.8   4  108  93 3.85 2.32 18.6  1  1    4
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.21 19.4  1  0    3
#> Hornet Sportabout 18.7   8  360 175 3.15 3.44 17.0  0  0    3
```

如果更喜欢用表格来显示数据，那么你可以使用 `knitr::kable` 函数。以下代码可以生成表 20-1：

```
knitr::kable(
  mtcars[1:5, ],
  caption = "A knitr kable."
)
```

表20-1：knitr表格

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.62	16.5	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.88	17.0	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.32	18.6	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.21	19.4	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.0	0	0	3	2

使用 `?knitr::kable` 阅读相关文档，学习一下定制表格的其他方式。如果想要进行更加深入的定制，可以学习一下 xtable、stargazer、pander、tables 和 ascii 包，每个包都提供了一套根据 R 代码生成格式化表格的工具。

同样，很多选项可以控制在最终报告中嵌入图形的方式。21.7 节将继续介绍这些选项。

20.4.4 缓存

一般来说，R Markdown 在每次生成文档时都是完全从头开始的。这对于文档的可重复性非常重要，因为这样可以确保不漏掉代码中的每一步重要计算。但是，如果有些计算需要

花费大量时间，那么每次重新生成文档都会是一个非常痛苦的过程。我们对这个问题的解决方案是使用 `cache = TRUE`。当使用这个选项时，R Markdown 会将代码段输出保存在磁盘上一个具有特殊名称的文件中。在此后的运行中，knitr 会检查代码是否进行了修改，如果没有修改，则继续使用缓存结果。

必须谨慎使用这种缓存机制，因为 knitr 在默认情况下只检查代码，不检查代码的依赖关系。例如，以下的 `processed_data` 代码段依赖于 `raw_data` 代码段：

````
```{r raw_data}
rawdata <- readr::read_csv("a_very_large_file.csv")
```

```{r processed_data, cached = TRUE}
processed_data <- rawdata %>%
  filter(!is.na(import_var)) %>%
  mutate(new_variable = complicated_transformation(x, y, z))
```
````

缓存 `processed_data` 代码段意味着，只在 dplyr 管道操作被修改时才重新运行代码，但如果对 `read_csv()` 函数的调用被修改了，`processed_data` 是不会重新运行的。为了解决这个问题，可以设置 `dependson` 代码段选项：

````
```{r processed_data, cached = TRUE, dependson = "raw_data"}
processed_data <- rawdata %>%
  filter(!is.na(import_var)) %>%
  mutate(new_variable = complicated_transformation(x, y, z))
```
````

`dependson` 应该包含每个代码段的一个字符向量，其中包括缓存代码段依赖的所有代码段。只要 knitr 检测到某个依赖代码段被修改了，就会重新运行缓存代码段以更新结果。

注意，如果 a_very_large_file.csv 文件被修改了，代码段是不会进行更新的，因为 knitr 缓存机制只跟踪 .Rmd 文件内部的变化。如果想要跟踪外部文件的变化，可以使用 `cache.extra` 选项。这是一个非常霸道的 R 表达式，只要其内容发生变化，缓存就会失效。我们可以同时使用 `file.info()` 函数，它可以返回大量关于文件的信息，其中包括最后修改的时间。因此可以将代码修改如下：

````
```{r raw_data, cache.extra = file.info("a_very_large_file.csv")}
rawdata <- readr::read_csv("a_very_large_file.csv")
```
````

因为缓存策略会逐渐变得复杂，所以应该定期使用 `knitr::clean_cache()` 命令清除所有缓存。

我们遵从了 David Robinson 给出的命名代码段的建议，即使用代码段生成的主要对象来命名代码段，这样可以使 `dependson` 设置更容易理解。

20.4.5 全局选项

当越来越多地使用 knitr 时，你会发现一些默认的代码段设置并不符合你的需要，因此想要

修改它们。你可以在代码段中调用 knitr::opts_chunk$set() 函数来修改选项。例如，在编写书籍和教程时，我们会设置如下：

```
knitr::opts_chunk$set(
  comment = "#>",
  collapse = TRUE
)
```

这样可以使用我们更喜欢的注释格式，并确保代码和输出能够紧密地组织在一起。另一方面，如果想要准备一份报告，则应该设置如下：

```
knitr::opts_chunk$set(
  echo = FALSE
)
```

这样可以默认隐藏代码，（使用 echo = TRUE）只显示你特意想展示出来的代码段。你还可以设置 message = FALSE 和 warning = FALSE，但这会使得调试变得更加困难，因为你在最终文档中看不到任何系统消息。

20.4.6 内联代码

将 R 代码嵌入 R Markdown 文档的另一种方法是：使用 r 直接将代码嵌入文档。如果想在文本中加入数据属性，那么这种方法是非常奏效的。例如，我们在本章开头所用的示例文档中有以下文本。

> 这份数据中包含了 r nrow(diamonds) 颗钻石的信息。其中只有 r nrow(diamonds) - nrow(smaller) 颗钻石大于 2.5 克拉。其余钻石的分布如下所示。

当生成最终报告时，knitr 会计算出 R 代码的结果，并插入文本。

> 这份数据中包含了 53 940 颗钻石的信息。其中只有 126 颗钻石大于 2.5 克拉。其余钻石的分布如下所示。

当向文本中插入数值时，format() 函数非常有用，你可以使用这个函数来设置 digits 的数值，避免打印出不必要的小数位，还可以设置 big.mark，从而使得数值更加易读。我们经常使用一个辅助函数来同时完成这两种设置：

```
comma <- function(x) format(x, digits = 2, big.mark = ",")
comma(3452345)
#> [1] "3,452,345"
comma(.12358124331)
#> [1] "0.12"
```

20.4.7 练习

(1) 为本章开头的 .Rmd 文件添加一节内容，研究钻石大小是如何随着切割、颜色和纯净度的不同而变化的。假设你要为不懂得 R 语言的人撰写一份报告，不要在每个代码段上设置 echo = FALSE，而是设置全局选项。

(2) 从 https://github.com/hadley/r4ds/tree/master/rmarkdown 下载 diamond-sizes.Rmd 文件，为其添加一节内容，描述最大的 20 颗钻石，其中要包括一个能显示出它们最重要特性的表格。

(3) 修改 diamond-sizes.Rmd 文件，使用 `comma()` 函数生成格式美观的输出，输出中还要包括大于 2.5 克拉的钻石的百分比。

(4) 建立一个代码段网络，其中 `d` 依赖于 `c` 和 `b`，`b` 和 `c` 都依赖于 `a`。每个代码段都输出 `lubridate::now()`，设置 `cache = TRUE`，并验证自己对缓存机制的理解。

20.5 排错

为 R Markdown 文件排查错误是比较困难的，因为我们不再拥有交互的 R 语言环境了，因此需要学习一些新的技巧。首先，我们必须尽力在交互会话中重现问题。重新启动 R，然后执行 Run all chunks（可以使用 Code 菜单，在 Run Region 菜单项下执行 Run All，也可以使用组合键 Ctrl+Alt+R）。如果你运气不错，就可以重现问题。这种交互方式可以帮助找出问题所在。

如果这样行不通，那一定是因为你的交互式环境和 R Markdown 环境之间有差别，你应该系统地检查一下选项设置。最常见的差别在于工作目录，R Markdown 的工作目录就是文件所在的目录。你可以在代码段中加入 `getwd()` 函数，查看工作目录是否符合预期。

接下来，尽力思考可能引起问题的所有因素。你需要系统地检查一下这些因素在 R 会话和 R Markdown 会话中是否完全一致。最简单的做法就是在引起问题的代码段中设置 `error = TRUE`，然后使用 `print()` 和 `str()` 函数检查这些因素是否与你预想的一致。

20.6 YAML文件头

通过调整 YAML 文件头中的参数，你可以控制很多其他的“全文档”设置。你或许很想知道 YAML 的意义，它代表的是**仍是一种标记语言**（yet another markup language）。YAML 设计用于表示容易被人类读写的层次化数据。R Markdown 使用 YAML 来控制很多输出细节。接下来将介绍两种 YAML 文件头：文档参数和参考文献。

20.6.1 文档参数

R Markdown 文件中可以包含在生成报告时进行设置的一个或多个参数。如果想要重新生成报告，但对多个关键输入使用与原来不同的值，那么这些参数就非常有用了。例如，你可以生成不同部门的销售报告、不同学生的考试结果，等等。如果想要声明一个或多个参数，可以使用 `params` 域。

以下示例使用了 `my_class` 参数来确定展示哪一类汽车：

```
---
output: html_document
params:
  my_class: "suv"
```

````
---

```{r setup, include = FALSE}
library(ggplot2)
library(dplyr)

class <- mpg %>% filter(class == params$my_class)
```

# Fuel economy for `r params$my_class`s

```{r, message = FALSE}
ggplot(class, aes(displ, hwy)) +
  geom_point() +
  geom_smooth(se = FALSE)
```
````

正如你看到的，在代码段中，参数设置在名为 `params` 的只读列表中。

你可以将原子向量直接写在 YAML 文件头中。在参数值前面加一个 `!r` 即可运行任意 R 表达式。这是设置日期 / 时间参数的一种绝好方法：

```
params:
  start: !r lubridate::ymd("2015-01-01")
  snapshot: !r lubridate::ymd_hms("2015-01-01 12:30:00")
```

在 RStudio 中，你可以点击 Knit 下拉菜单中的 Knit with Parameters 菜单项来完成设置参数、生成报告和预览报告等工作。这是用户友好型的一个步骤。你可以通过设置文件头中的其他选项来定制对话框，详细信息参见 https://rmarkdown.rstudio.com/developer_parameterized_reports.html。

另外，如果需要生成多个这种带参数的报告，可以使用一个 `params` 列表来调用 `rmarkdown::render()` 函数：

```
rmarkdown::render(
  "fuel-economy.Rmd",
  params = list(my_class = "suv")
)
```

这个函数与 `purrr::pwalk()` 函数组合使用时功能非常强大。以下示例为 `mpg` 数据集中的每个 `class` 值都创建了一份报告。首先，我们创建一个数据框，其中每行都代表一类汽车，并给出了报告的文件名 `filename` 以及应该设定的参数 `params`：

```
reports <- tibble(
  class = unique(mpg$class),
  filename = stringr::str_c("fuel-economy-", class, ".html"),
  params = purrr::map(class, ~ list(my_class = .))
)
reports
#> # A tibble: 7 × 3
#>     class                  filename     params
#>     <chr>                     <chr>     <list>
#> 1 compact fuel-economy-compact.html <list [1]>
```

```
#> 2 midsize fuel-economy-midsize.html <list [1]>
#> 3     suv     fuel-economy-suv.html <list [1]>
#> 4 2seater fuel-economy-2seater.html <list [1]>
#> 5 minivan fuel-economy-minivan.html <list [1]>
#> 6  pickup  fuel-economy-pickup.html <list [1]>
#> # ... with 1 more rows
```

然后我们将列名与 `render()` 函数的参数名进行匹配，并使用 purrr 包的**并行**游走函数为每一行调用一次 `render()`：

```
reports %>%
  select(output_file = filename, params) %>%
  purrr::pwalk(rmarkdown::render, input = "fuel-economy.Rmd")
```

20.6.2 参考文献与引用

pandoc 可以自动生成引用和各种风格的参考文献。要想使用这个功能，需要在文件头中的 `bibliography` 域中设定一个参考文献文件。这个域应该包含一个从 .Rmd 文件目录到参考文献文件目录的路径：

```
bibliography: rmarkdown.bib
```

你可以使用多种常用的参考文献格式，如 BibLaTeX、BibTeX、endnote 和 medline。

要想在 .Rmd 文件中创建一个引用，你可以使用一个**由 @ 和引用标识符**组成的关键词，其中引用标识符来自于参考文献文件。接着需要将引用放在方括号中。以下是一些示例：

```
Separate multiple citations with a `;`:
Blah blah [@smith04; @doe99].

You can add arbitrary comments inside the square brackets:
Blah blah [see @doe99, pp. 33-35; also @smith04, ch. 1].

Remove the square brackets to create an in-text citation:
@smith04 says blah, or @smith04 [p. 33] says blah.

Add a `-` before the citation to suppress the author's name:
Smith says blah [-@smith04].
```

在生成最终文件时，R Markdown 会创建参考文献并将其追加到文件的最后。参考文献中会包含来自于参考文献文件中被引用的所有文献，但不会包含小节标题。因此，通常的做法是在文件的末尾为参考文献加上一个小节标题，如 `# References` 或者 `# Bibliography`。

你可以在 `csl` 域中引用一个 CSL（citation style language，引用样式语言）文件来改变引用和参考文献的风格：

```
bibliography: rmarkdown.bib
csl: apa.csl
```

对于参考文献域，你的 CSL 文件应该包含一个到参考文献文件的路径。在这个示例中，我们假设 CSL 文件和 .Rmd 文件位于同一目录下。你可以在 http://github.com/citation-style-language/styles 中找到控制常用参考文件风格的 CSL 样式文件。

20.7 更多学习资源

R Markdown 还不算很成熟，正处于快速发展阶段。如果想要关注这项技术的最新发展，可以访问 R Markdown 官方网址。

本章没有介绍的两个重要主题是：协同工作以及与他人精确沟通自己想法的具体做法。协同工作是现代数据科学中极其重要的一部分，使用像 Git 和 GitHub 这样的版本控制工具可以让你的工作变得更加轻松。以下推荐了学习 Git 的两种免费资源。

- “Happy Git with R”：面向 R 语言用户的用户友好型 Git 与 GitHub 入门简介，Jenny Bryan 著。这本书有免费的在线版本。
- 《R 包开发》[1] 中的“Git 和 GitHub”一章，Hadley 著。你也可以在线免费阅读这部分内容。

要想明确无误地与他人沟通分析结果，应该使用何种写作方法与写作技巧也是本章没有讲到的内容。为了提高写作水平，我们强烈推荐你要么阅读一下 Joseph M. Williams 和 Joseph Bizup 所著的 *Style: Lessons in Clarity*，要么阅读一下 George Gopen 所著的 *The Sense of Structure: Writing from the Reader's Perspective*。这两本书都可以帮助你理解句子和段落的结构，并掌握使得文章更加清晰易读的方法。（这两本书都非常昂贵，但因为很多英文课堂都使用这两本书作为教材，所以市面上有大量的便宜二手书。）George Gopen 还撰写了很多关于写作的文章，虽然这些文章的目标读者是律师，但几乎所有内容都适合数据科学家。

注 1：此书已由人民邮电出版社出版，详见 http://www.ituring.com.cn/book/1688。——编者注

第 21 章

使用ggplot2进行图形化沟通

21.1 简介

第 5 章中介绍了使用图形进行**数据探索**的方法。在制作探索性图形时，我们对图形要显示哪些变量已经了然于心。我们出于某种目的制作了一幅图形，快速地对其进行检视，然后继续下一幅。大多数分析过程都会生成几十或几百幅图形，其中大部分都会被立刻弃之不用。

理解了数据后，就需要与人们**沟通**我们的想法。听众很可能不具备我们所拥有的背景知识，也不会对数据投入太多精力。为了帮助人们快速建立一个比较好的数据思维模型，我们需要付出相当大的努力，尽可能让图形通俗易懂。在本章中，我们将学习 ggplot2 中提供的一些方法来完成这个任务。

本章将重点介绍创建良好图形所需的方法。我们假设你已经明确了需求，只是想知道如何来做。因此，强烈建议你在学习本章的同时，再阅读一本比较好的可视化书籍。我们特别喜欢 Albert Cairo 的著作 *The Truthful Art*，这本书讲述的不是创建可视化图形的技术，而是重点介绍在创建有效图形时需要考虑的因素。

准备工作

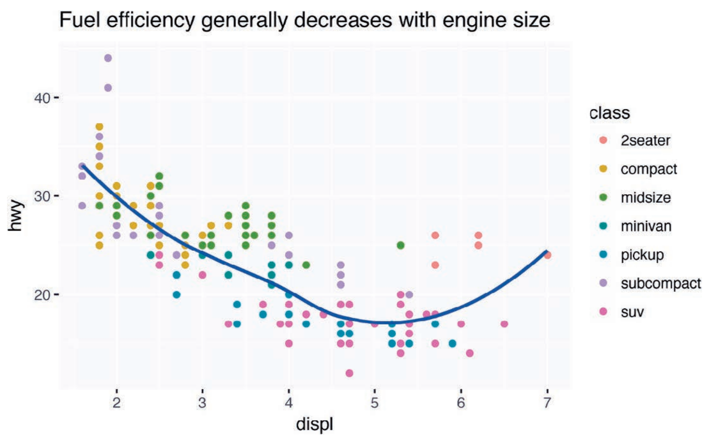
本章将再次重点介绍 ggplot2。我们还会使用 dplyr 包的一些功能来进行数据处理，并介绍 ggplot2 的几个扩展包，其中包括 ggrepel 和 viridis。我们不用加载这些扩展包，而是选择使用 :: 表示法显式地引用其中的函数，这样可以更清楚地分辨出哪些函数是 ggplot2 的内置函数，哪些函数来自于其他扩展包。别忘了，如果还没有安装这些扩展包，那么你需要使用 `install.packages()` 函数进行安装。

```
library(tidyverse)
```

21.2 标签

要想将探索性图形转换为解释性图形，最容易的方法就是加上一些合适的标签。我们可以使用 labs() 函数来添加标签，以下示例为图形添加了一个标题：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
   title = paste(
     "Fuel efficiency generally decreases with"
     "engine size"
  )
```

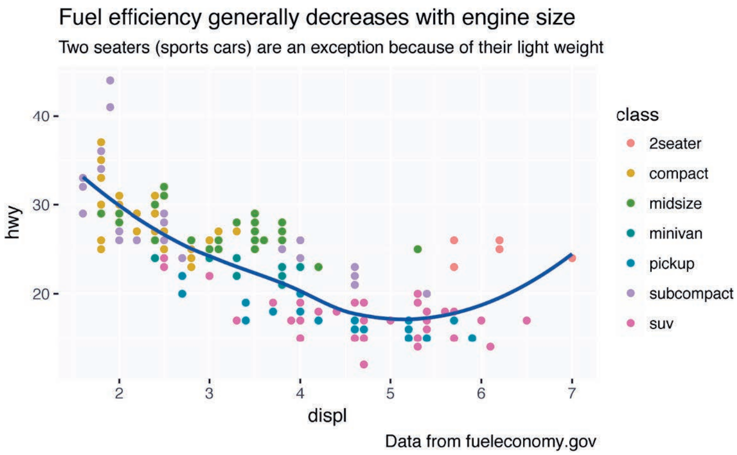


使用图形标题的目的是概括主要成果。尽量不要使用那些只对图形进行描述的标题，如“发动机排量与燃油效率散点图”。

如果想要添加更多文本，ggplot2 2.2.0 及其更高版本（当你读到本书时，更高版本肯定已经可用了）中还提供了另外 2 个非常实用的标签。

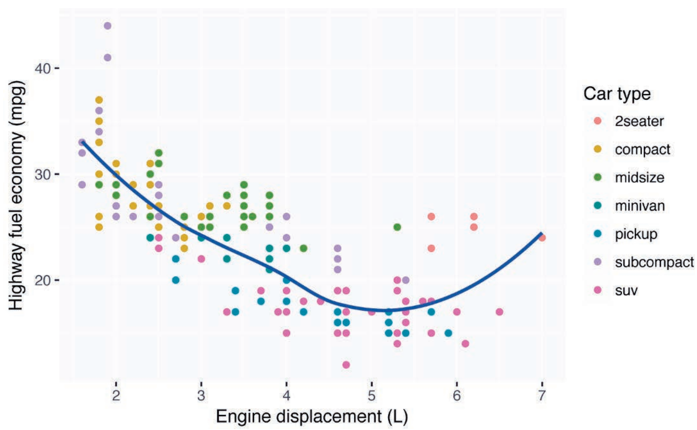
- subtitle 可以在标题下以更小的字体添加更多附加信息。
- caption 可以在图形右下角添加文本，常用于描述数据来源：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
    title = paste(
      "Fuel efficiency generally decreases with"
      "engine size",
    )
    subtitle = paste(
      "Two seaters (sports cars) are an exception"
      "because of their light weight",
    )
    caption = "Data from fueleconomy.gov"
  )
```



我们还可以使用 labs() 函数来替换坐标轴和图例中的标题。将简短的变量名称替换为更详细的描述并加上单位，是一种非常好的做法：

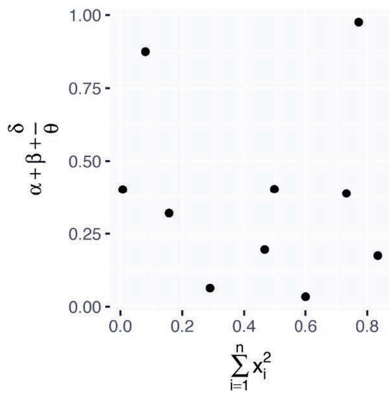
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
    x = "Engine displacement (L)",
    y = "Highway fuel economy (mpg)",
    colour = "Car type"
  )
```



还可以在标题中使用数学公式代替字符串文本，用 quote() 函数代替 ""，再使用 ?plotmath 命令查看可用选项：

```
df <- tibble(
  x = runif(10),
  y = runif(10)
)
ggplot(df, aes(x, y)) +
```

```
  geom_point() +
  labs(
    x = quote(sum(x[i] ^ 2, i == 1, n)),
    y = quote(alpha + beta + frac(delta, theta))
  )
```



练习

(1) 使用燃油效率数据创建一幅图形，自定义 `title`、`subtitle`、`caption`、`x`、`y` 和 `colour` 标签。

(2) `geom_smooth()` 函数具有一定的误导性，因为对于大排量引擎来说，由于包括了车身重量很轻的大引擎跑车，hwy 数据有些偏高。选择合适的建模方法拟合一个更好的模型，并用图形表示出来。

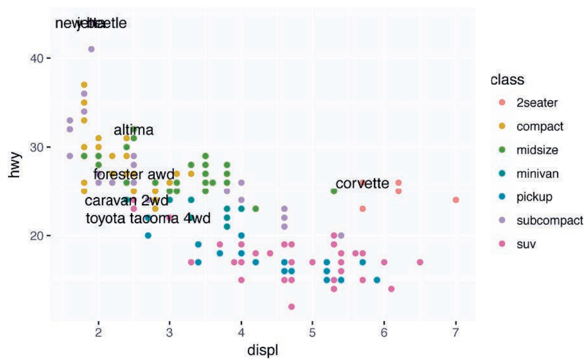
(3) 选择一个你以前制作的探索性图形，添加各种信息丰富的标题，以便更易理解。

21.3 注释

除了为图形中的主要部分添加标签，我们还经常需要为单个观测或分组观测添加标签。可供我们使用的第一个工具是 `geom_text()` 函数，其用法基本与 `geom_point()` 函数相同，但具有一个额外的图形属性：`label`。这使得我们可以向图形中添加文本标签。

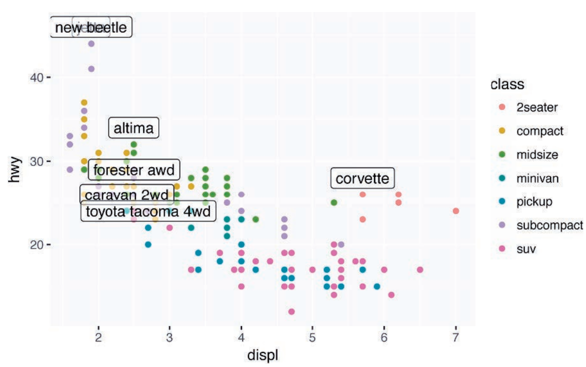
可以使用 2 种方式来提供标签。首先，可以使用 tibble。以下图形的实际用处不大，但可以说明一种非常有用的方法：先使用 dplyr 选取出每类汽车中效率最高的型号，然后在图形中标记出来：

```
best_in_class <- mpg %>%
  group_by(class) %>%
  filter(row_number(desc(hwy)) == 1)
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_text(aes(label = model), data = best_in_class)
```



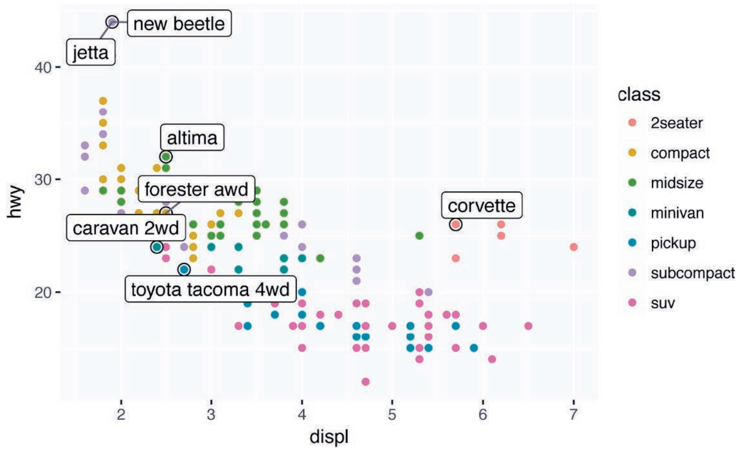
这幅图不太好看，因为标签不但彼此重叠，还与数据点混在一起。改进这个问题的方法是换用 `geom_label()` 函数，它可以为文本加上方框。我们还可以使用 `nudge_y` 参数让标签位于相应数据点的正上方：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_label(
    aes(label = model),
    data = best_in_class,
    nudge_y = 2,
    alpha = 0.5
  )
```



这样效果更好一些。但如果仔细查看左上角，你就会发现有 2 个标签几乎完全重叠。发生这种情况的原因是，对于小型车和微型车中燃油效率最高的汽车来说，它们的公路里程数和引擎排量是完全相同的。我们无法通过对每个标签进行转换来解决这个问题，但是可以使用由 Kamil Slowikowski 开发的 ggrepel 包。这个包非常有用，可以自动调整标签的位置，使它们免于重叠：

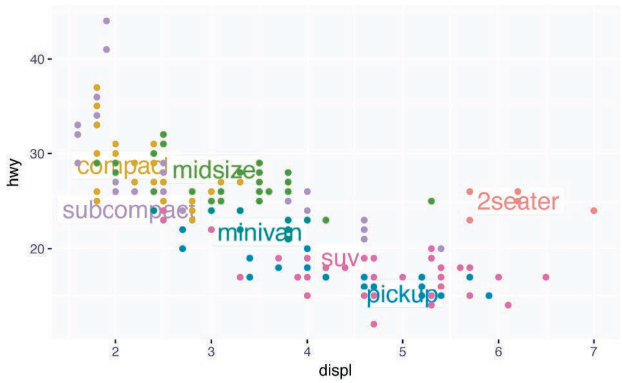
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_point(size = 3, shape = 1, data = best_in_class) +
  ggrepel::geom_label_repel(
    aes(label = model),
    data = best_in_class
  )
```



注意，我们还做了另一项很贴心的改动：添加了一个图层，用较大的空心圆来强调添加了标签的数据点。

有时你可以通过同样的方式将标签直接放在图形上，以替代图例。这种方式不适合本示例中的图形，但有时效果还是不错的。（theme(legend.positon = "none") 可以不显示图例，稍后我们将进行更多讨论。）

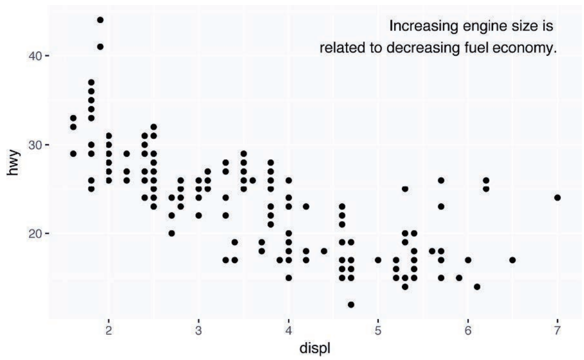
```
class_avg <- mpg %>%
  group_by(class) %>%
  summarize(
    displ = median(displ),
    hwy = median(hwy)
  )
ggplot(mpg, aes(displ, hwy, color = class)) +
  ggrepel::geom_label_repel(aes(label = class),
    data = class_avg,
    size = 6,
    label.size = 0,
    segment.color = NA
  ) +
  geom_point() +
  theme(legend.position = "none")
```



其次，即使只想向图中添加唯一的一个标签，也需要创建一个数据框。通常来说，你会希望标签在图的角落，因此，应该使用 summarize() 函数计算出 x 和 y 的最大值，并保存在数据框中：

```
label <- mpg %>%
  summarize(
    displ = max(displ),
    hwy = max(hwy),
    label = paste(
      "Increasing engine size is \nrelated to"
      "decreasing fuel economy."
    )
  )

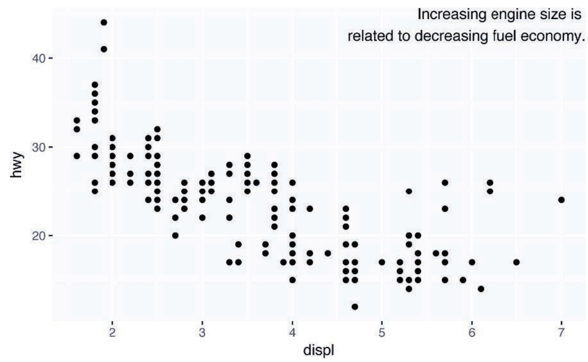
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_text(
    aes(label = label),
    data = label,
    vjust = "top",
    hjust = "right"
  )
```



如果想让标签紧贴着图形的边界，那么你可以使用 +Inf 和 -Inf 值。这样我们就无须再从 mpg 数据集中计算位置了，因此可以使用 tibble() 函数直接创建数据框：

```
label <- tibble(
  displ = Inf,
  hwy = Inf,
  label = paste(
    "Increasing engine size is \nrelated to"
    "decreasing fuel economy."
  )
)

ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_text(
    aes(label = label),
    data = label,
    vjust = "top",
    hjust = "right"
  )
```



在这些示例中，我们使用 "\n" 手动为标签文本换行。另一种方法是使用 stringr::str_wrap() 函数来自动换行，此时需要给出每行的字符数：

```
"Increasing engine size related to decreasing fuel economy." %>%
  stringr::str_wrap(width = 40) %>%
  writeLines()
#> Increasing engine size is related to
#> decreasing fuel economy.
```

注意，hjust 和 vjust 是用于控制标签的对齐方式。图 21-1 给出了所有 9 种可能的组合。

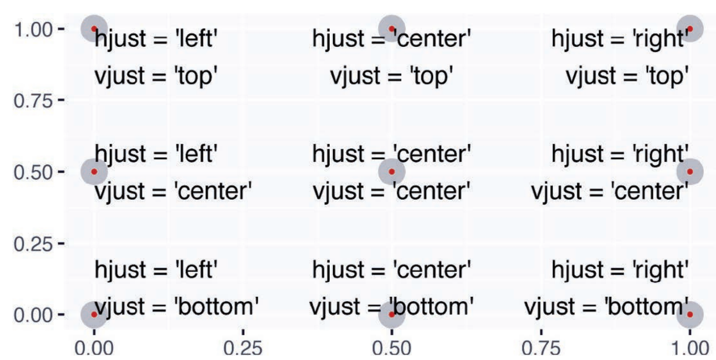


图 21-1：hjust 和 vjust 的所有 9 种组合

记住，除了 geom_text()，ggplot2 中还有很多其他函数可以在图形中添加注释，具体如下。

- 可以使用 geom_hline() 和 geom_vline() 函数添加参考线。我们经常使用加粗（size = 2）和白色（color = white）的直线作为参考线，并将它们绘制在基本数据层的下面。这样的参考线既清晰可见，又不至于喧宾夺主，影响我们查看数据。
- 可以使用 geom_rect() 函数在我们感兴趣的数据点周围绘制一个矩形。矩形的边界由图形属性 xmin、xmax、ymin 和 ymax 确定。
- 可以使用 geom_segment() 函数及 arrow 参数绘制箭头，指向需要关注的数据点。使用图形属性 x 和 y 来定义开始位置，使用 xend 和 yend 来定义结束位置。

使用这些技术的唯一限制就是你的想象力（以及调整注释位置让图形更美观所需要的耐心）！

练习

(1) 使用 `geom_text()` 函数和无穷大参数值将文本标签放置在图形的 4 个角落。

(2) 阅读 `annotate()` 函数的文档。不创建 tibble 的情况下，如何使用这个函数为图形添加一个文本标签？

(3) 使用 `geom_text()` 创建的标签与分面是如何相互影响的？如何在一个分面中添加标签？如何在每个分面中添加不同标签？（提示：思考一下基础数据。）

(4) `geom_label()` 函数的哪个参数可以控制背景框的外观？

(5) `arrow()` 函数的 4 个参数是什么？它们有什么作用？创建一组图形来展示这几个参数中最重要的几个选项。

21.4 标度

使得图表更适合沟通的第三种方法是调整标度。标度控制着从数据值到图形属性的映射，它可以将数据转换为视觉上可以感知的东西。一般情况下，ggplot2 会自动向图表中添加标度。例如，如果输入以下代码：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class))
```

ggplot2 会自动在后台为代码添加默认标度：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  scale_x_continuous() +
  scale_y_continuous() +
  scale_color_discrete()
```

注意标度的命名模式：`scale_` 后面是图形属性的名称，然后是 `_`，再然后是标度的名称。默认情况下，标度是以变量最可能的类型来命名的：连续型、离散型、日期时间型或日期型。还有很多非默认标度，我们会在后面的内容中进行介绍。

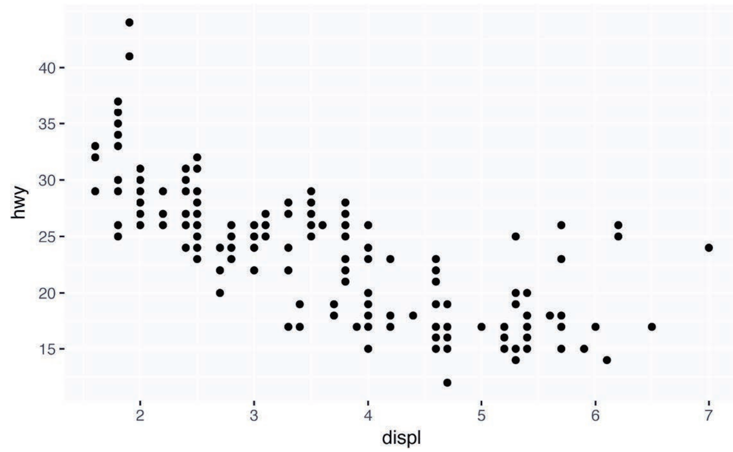
默认标度是精心选择的，适合多种输入。但基于以下两种原因，你可以不使用默认标度。

- 你或许要对默认标度的一些参数进行调整。例如，当想要修改坐标轴刻度或图例中的项目标签时，就需要进行这些调整。
- 你或许想要整体替换默认标度，从而使用一种完全不同的算法。因为你对数据更加了解，所以使用与默认方式不同的标度通常能达到更好的效果。

21.4.1 坐标轴刻度与图例项目

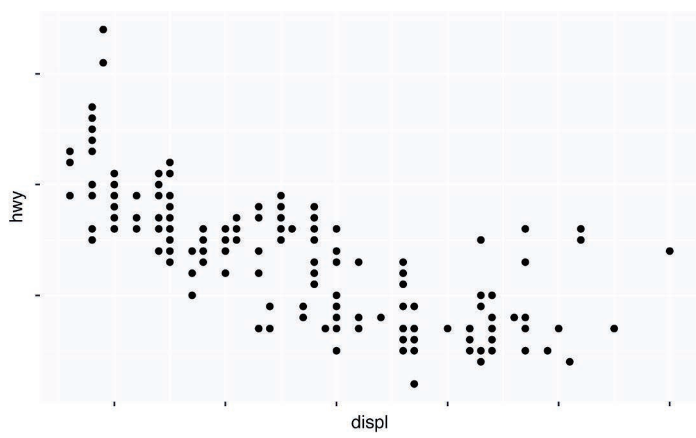
影响坐标轴刻度与图例项目外观的主要参数有两个：`breaks` 和 `labels`。`breaks` 控制坐标轴刻度的位置，以及与图例项目相关的数值显示。`labels` 控制与每个坐标轴刻度或图例项目相关的文本标签。`breaks` 的最常见用途是替换默认的刻度：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_y_continuous(breaks = seq(15, 40, by = 5))
```



你可以用同样的方式来使用 labels（与 breaks 长度相同的字符向量）。你还可以将其设置为 NULL，这样可以不显示刻度标签，对于地图或不适合展示数值的图表来说，这种方式是非常有用的：

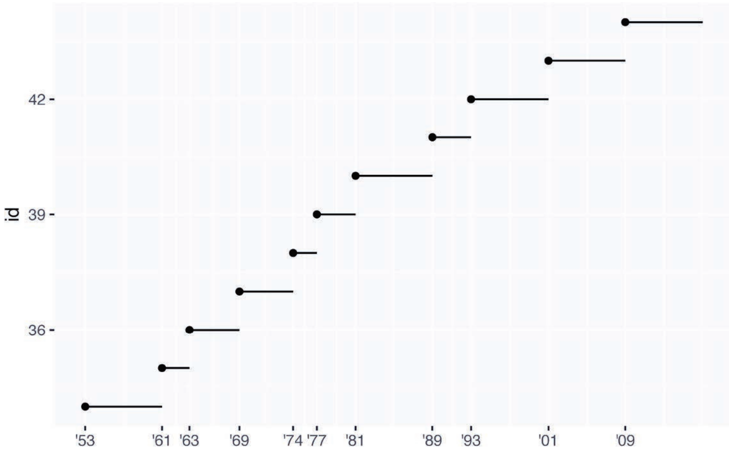
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_x_continuous(labels = NULL) +
  scale_y_continuous(labels = NULL)
```



你还可以使用 breaks 和 labels 控制图例的外观。坐标轴和图例统称为**引导元素**。坐标轴用于表示 x 和 y 图形属性；图例则用于表示其他的引导性信息。

需要使用 breaks 的另一种情况是，数据点相对较少，而你又想要强调观测的确切位置。例如，以下图形展示了每位美国总统任期的开始时间和结束时间：

```
presidential %>%
  mutate(id = 33 + row_number()) %>%
  ggplot(aes(start, id)) +
    geom_point() +
    geom_segment(aes(xend = end, yend = id)) +
    scale_x_date(
      NULL,
      breaks = presidential$start,
      date_labels = "'%y"
    )
```



注意，对于日期型和日期时间型标度来说，刻度和标签的格式有一些不同。

- date_labels 接受一个格式说明，说明的形式与 parse_datetime() 函数中的相同。
- date_breaks（这个示例中没有出现）则使用类似“2 天”或“1 个月”这样的字符串。

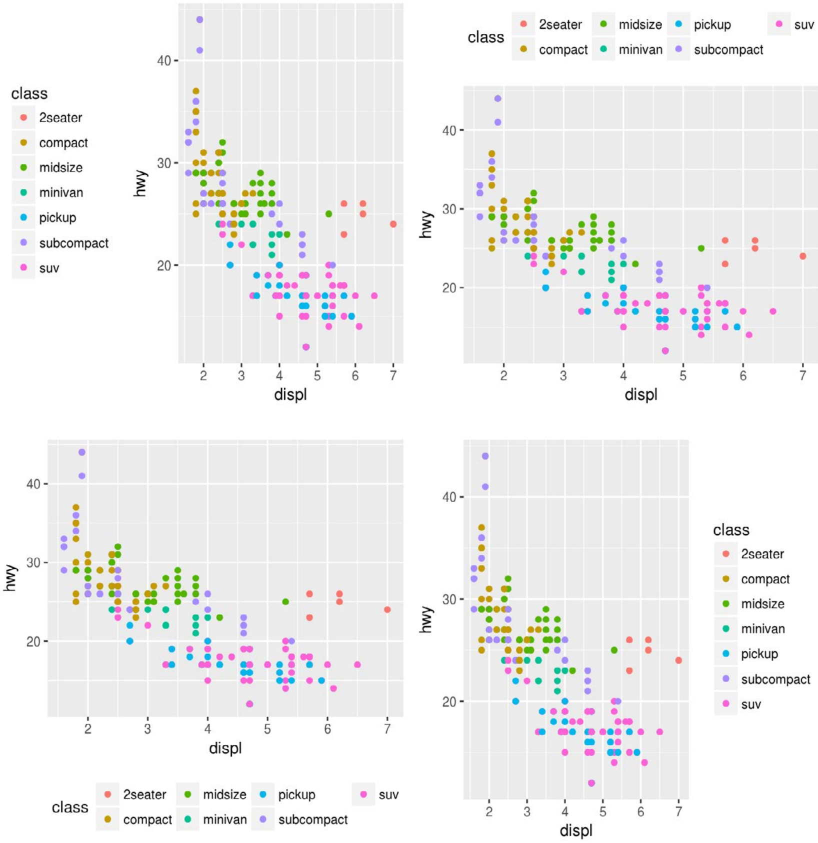
21.4.2 图例布局

breaks 和 labels 最常用于调整坐标轴。然而，它们也可以用于调整图例，我们再介绍几种你很可能会用到的技术。

如果要控制图例的整体位置，你需要使用 theme() 函数进行设置。本章末尾将介绍关于主题的知识，简而言之，主题的作用就是控制图形中与数据无关的部分。主题设置中的 legend.position 可以控制图例的位置：

```
base <- ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class))

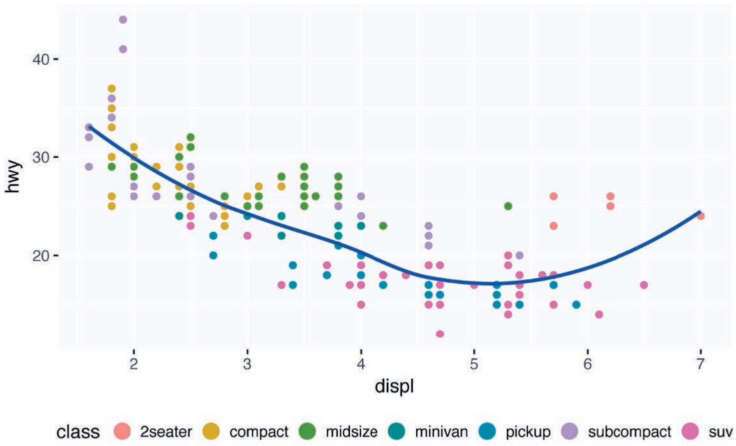
base + theme(legend.position = "left")
base + theme(legend.position = "top")
base + theme(legend.position = "bottom")
base + theme(legend.position = "right") # 默认设置
```



你还可以使用 legend.positon = "none" 来取消整个图例的显示。

要想控制单个图例的显示，可以配合 guide_legend() 或 guide_colorbar() 函数来使用 guides() 函数。在以下示例中，我们给出了两个重要设置：使用 nrow 设定图例项目的行数，并覆盖一个图形属性，以便数据点更大一些。如果想要在一张图表上使用较低的 alpha 值显示多个数据点，那么这些设置尤为重要：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  theme(legend.position = "bottom") +
  guides(
    color = guide_legend(
      nrow = 1,
      override.aes = list(size = 4)
    )
  )
#> `geom_smooth()` using method = 'loess'
```



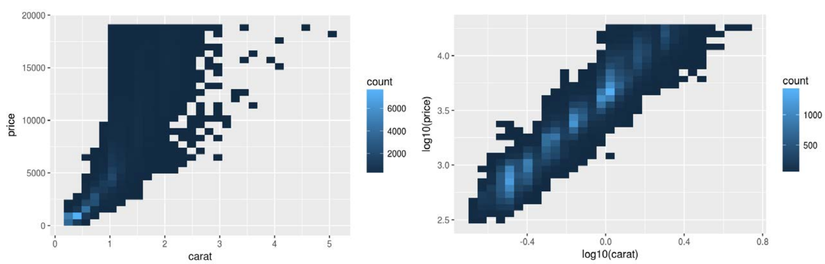
21.4.3　标度替换

除了对标度的细节略做调整，我们还可以替换整体标度。最经常进行替换的两种标度是连续型位置标度和颜色标度。好在其中原理适用于其他图形属性，因此一旦掌握了位置和颜色的替换方法，就可以迅速学会其他各种标度替换。

绘制出变量转换是非常有用的。例如，正如我们在 18.2 节中看到的，如果对 carat 和 price 进行对数转换，就更容易看出二者间的确切关系：

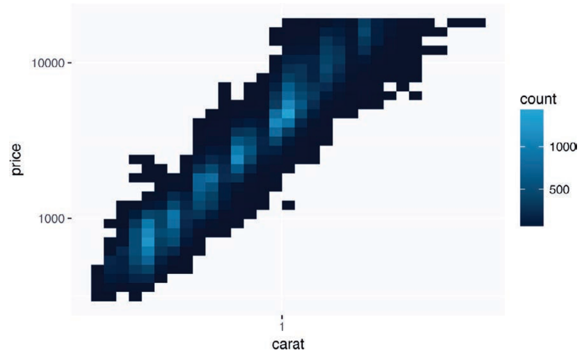
```
ggplot(diamonds, aes(carat, price)) +
  geom_bin2d()

ggplot(diamonds, aes(log10(carat), log10(price))) +
  geom_bin2d()
```



但这种变换的缺点是，因为坐标轴是以转换后的值来标记的，所以很难解释图表。除了在图形属性映射中进行转换，我们还可以使用标度进行转换。二者的视觉效果是一样的，只是进行了标度变换后，坐标轴还是以原始数据标度进行标记的：

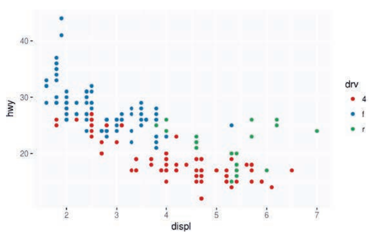
```
ggplot(diamonds, aes(carat, price)) +
  geom_bin2d() +
  scale_x_log10() +
  scale_y_log10()
```

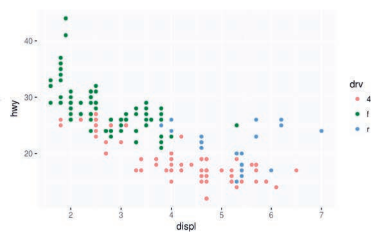


经常需要修改定制的另一种标度是颜色。默认分类标度以一种非常均匀的方式在色环上选择颜色。常用的另一种配色方式是使用 ColorBrewer 标度，经过手工调整后，这种方式更适合那些有色盲症的人。以下的两幅图非常相似，但是右边图中的红色和绿色的对比更加强烈，即使是患有红绿色盲症的人也可以区别出来：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv))

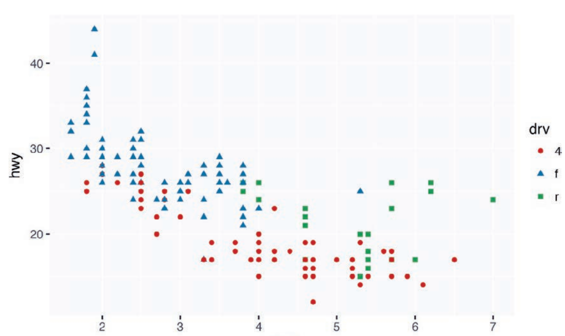
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv)) +
  scale_color_brewer(palette = "Set1")
```





别小看简单的技术。如果只有很少几种颜色，那么你完全可以再添加一种形状映射。虽然有些冗余，但这样可以确保图表在黑白方式下也是可以为人所理解的：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv, shape = drv)) +
  scale_color_brewer(palette = "Set1")
```



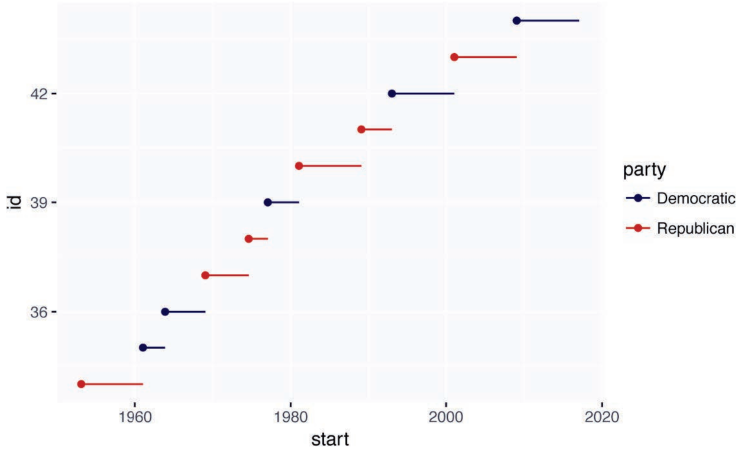
ColorBrewer 标度的在线文档地址是 http://colorbrew2.org，在 R 中可以通过 RColorBrewer 包实现，这个包的作者是 Erich Neuwirth。图 21-2 给出了所有调色板的完整列表。如果分类变量的值是有顺序的或者有一个“中间值”，那么上面的顺序调色板和下面的发散调色板就尤为重要。在使用 `cut()` 函数将连续型变量转换为分类变量时，这两种情况经常出现。



图 21-2：ColorBrewer 中的全部配色方案

如果预先确定了数据值和颜色间的映射，那么可以使用 scale_color_manual() 函数。例如，我们可以将总统的党派映射到颜色，使用红色表示共和党，蓝色表示民主党：

```
presidential %>%
  mutate(id = 33 + row_number()) %>%
  ggplot(aes(start, id, color = party)) +
    geom_point() +
    geom_segment(aes(xend = end, yend = id)) +
    scale_colour_manual(
      values = c(Republican = "red", Democratic = "blue")
    )
```

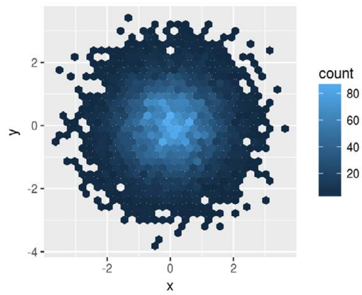


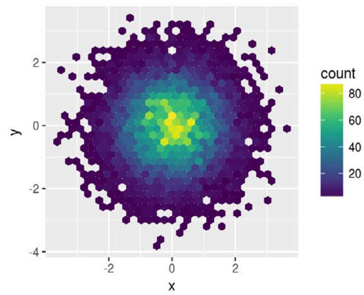
对于连续的颜色标度，我们可以使用内置函数 scale_color_gradient() 或 scale_fill_gradient() 来表示。如果想要表示发散性的颜色标度，可以使用 scale_color_gradient2() 函数，它可以使用正数和负数来表示不同的颜色。如果想要区分出位于平均值以上和以下的点，那么这个函数是非常合适的。

另一个可以选用的函数是由 viridis 包提供的 scale_color_viridis()，它是对 ColorBrewer 分类标度的一种连续模拟，包作者 Nathaniel Smith 和 Stéfan van der Walt 精心设计了一种具有优秀感知特性的连续型颜色模式。以下是来自于 viridis 使用指南中的一个示例：

```
df <- tibble(
  x = rnorm(10000),
  y = rnorm(10000)
)
ggplot(df, aes(x, y)) +
  geom_hex() +
  coord_fixed()
#> Loading required package: methods

ggplot(df, aes(x, y)) +
  geom_hex() +
  viridis::scale_fill_viridis() +
  coord_fixed()
```





注意，所有的颜色标度都可以分为两类：scale_color_x() 对应于 color 图形属性，scale_fill_x() 则对应于 fill 图形属性（颜色标度既可以使用美式英语，也可以使用英式英语）。

21.4.4　练习

(1) 为什么以下代码没有覆盖默认标度？

```
ggplot(df, aes(x, y)) +
  geom_hex() +
  scale_color_gradient(low = "white", high = "red") +
  coord_fixed()
```

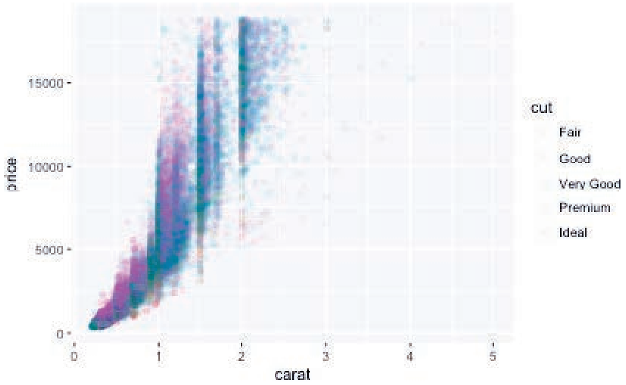
(2) 每个标度函数中的第一个参数的意义是什么？与 labs() 函数相比有什么不同？

(3) 按照以下要求修改 presidential 图形的显示。

a. 组合使用前面介绍过的两类颜色标度进行改进。
b. 美化 *y* 轴的显示。
c. 使用总统的姓名标注每个图形项目。
d. 添加各种说明性标签。
e. 每隔 4 年添加一个刻度线（这比想象的要难！）。

(4) 使用 override.aes 让下图中的图例更加一目了然。

```
ggplot(diamonds, aes(carat, price)) +
  geom_point(aes(color = cut), alpha = 1/20)
```



21.5 缩放

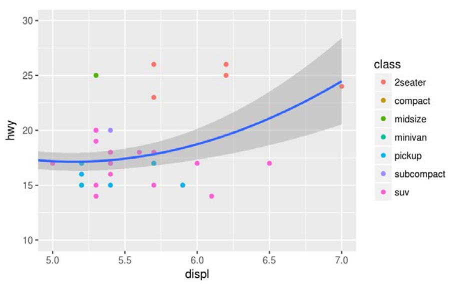
控制图形范围的方法有 3 种：

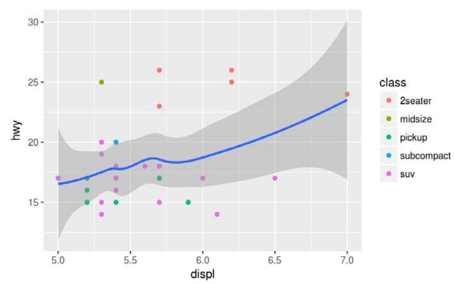
- 调整绘图所用数据；
- 设置标度范围；
- 在 `coord_cartesian()` 函数中设置 `xlim` 和 `ylim` 参数值。

如果想要放大图形的一片区域，最好使用 `coord_cartesian()` 函数。比较以下这两个图形：

```
ggplot(mpg, mapping = aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth() +
  coord_cartesian(xlim = c(5, 7), ylim = c(10, 30))

mpg %>%
  filter(displ >= 5, displ <= 7, hwy >= 10, hwy <= 30) %>%
  ggplot(aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth()
```



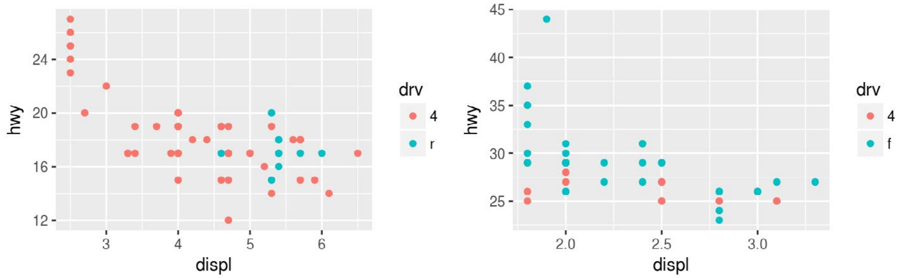


你也可以设置单个标度的范围。缩小标度范围的效果基本等同于对数据取子集。如果想要**扩大**图形范围，比如在不同图形间使用相同的标度，那么最好通过设置标度范围来实现。举例来说，如果提取两种不同类型汽车的数据，并想分别绘制出来，但因为这两份数据的 3 种标度（x 轴、y 轴和颜色图形属性）范围都不一样，所以很难进行比较：

```
suv <- mpg %>% filter(class == "suv")
compact <- mpg %>% filter(class == "compact")

ggplot(suv, aes(displ, hwy, color = drv)) +
  geom_point()

ggplot(compact, aes(displ, hwy, color = drv)) +
  geom_point()
```

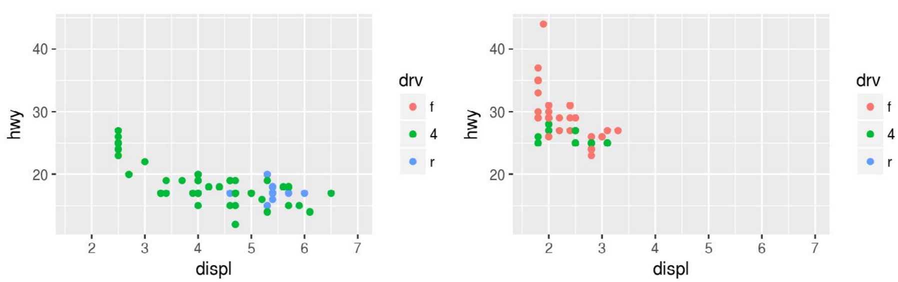


解决这个问题的一种方法是，先使用全部数据找出标度的 limits，然后在两张图形中使用同样的标度：

```
x_scale <- scale_x_continuous(limits = range(mpg$displ))
y_scale <- scale_y_continuous(limits = range(mpg$hwy))
col_scale <- scale_color_discrete(limits = unique(mpg$drv))

ggplot(suv, aes(displ, hwy, color = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale

ggplot(compact, aes(displ, hwy, color = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale
```

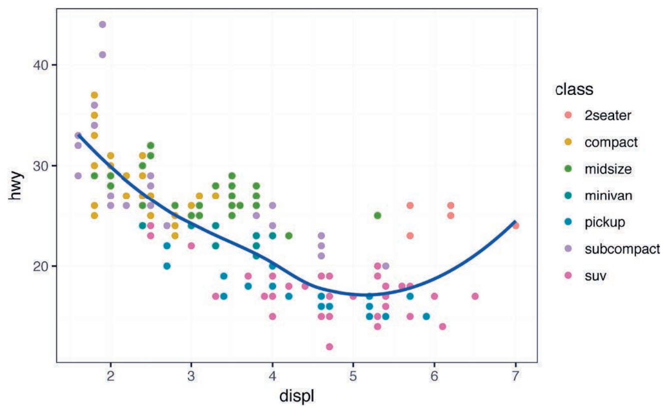


对于这个特定的示例，我们可以简单地使用分面来解决，但以上方法适用范围更广，例如，可以在报告中加入横跨多页的图形。

21.6 主题

最后，我们还可以使用主题来定制图形中的非数据元素：

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  theme_bw()
```



ggplot2 默认可以使用 8 种主题，如图 21-3 所示。在 Jeffrey Arnold 开发的 ggthemes 扩展包（https://github.com/jrnold/ggthemes）中，还可以使用更多主题。

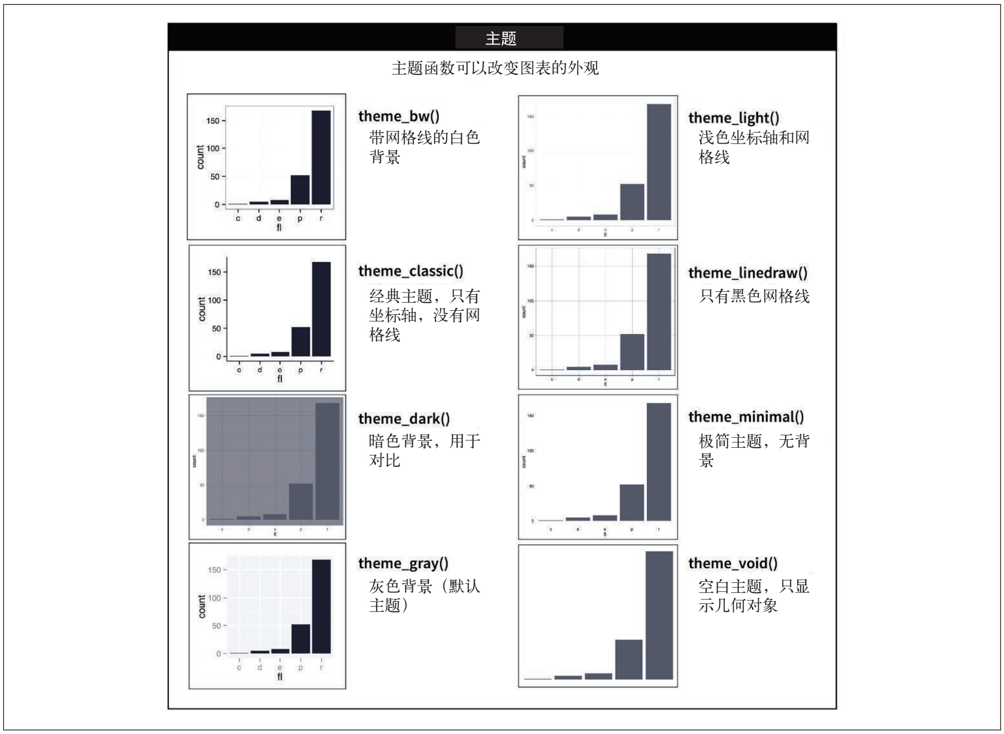


图 21-3：ggplot2 内置的 8 种主题

很多人会感到诧异，为什么默认主题要使用灰色背景。这是有意为之，因为这样可以在网格线可见的情况下更加突出数据。白色网格线既是可见的（这非常重要，因为它们非常有助于位置判定），又对视觉没有什么严重影响，我们完全可以对其视而不见。图表的灰色背景不像白色背景那么突兀，与文本印刷颜色非常相近，保证了图形与文档其他部分浑然

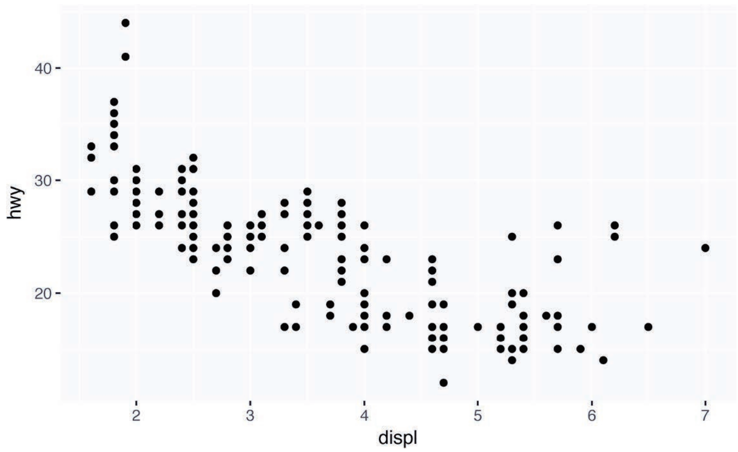
一体。最后，灰色背景可以创建一片连续的颜色区域，使得图形成为形象鲜明的一个独立视觉实体。

我们还可以控制每种主题中的独立成分，比如 *y* 轴字体的大小和颜色。遗憾的是，这些细节已经超出了本书的讨论范围，因此你需要学习关于 ggplot2 的图书（http://ggplot2.org/book/）才能掌握全部细节。如果为了满足公司或期刊的具体要求，你也可以创建属于自己的主题。

21.7　保存图形

要想将图形从 R 导入你的最终报告，主要有两种方法：`ggsave()` 和 knitr。`ggsave()` 可以将最近生成的图形保存到磁盘：

```
ggplot(mpg, aes(displ, hwy)) + geom_point()
```




```
ggsave("my-plot.pdf")
#> Saving 6 x 3.71 in image
```

如果没有指定 `width` 和 `height`，那么 `ggsave()` 就会使用当前绘图设备的尺寸。出于代码重用性的考虑，最好还是指定图形的这些参数。

然而，我们认为你通常会通过 R Markdown 生成最终报告，因此我们将重点介绍你在生成图形时应该了解的一些重要代码段选项。你可以通过说明文档来了解更多关于 `ggsave()` 的知识。

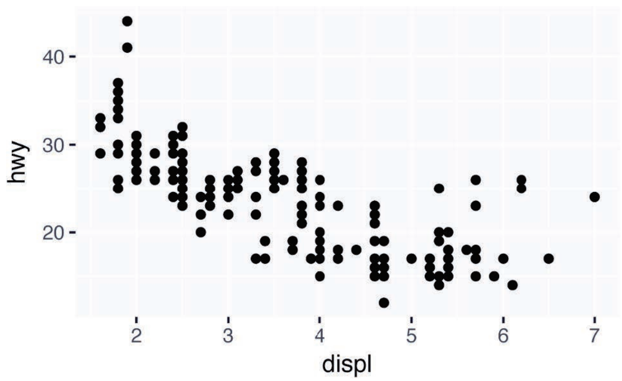
21.7.1　图形大小

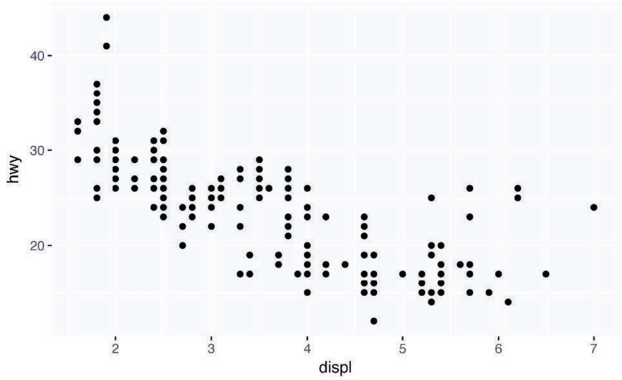
在 R Markdown 中，关于图形的最大问题是如何确定其大小和形状。控制图形大小的选项主要有 5 个：`fig.width`、`fig.height`、`fig.asp`、`out.width` 和 `out.height`。之所以说图形大小是一个难题，是因为图形大小有两种（R 生成的图形的大小，以及插入到输出文档中的图形的大小），而且指定图形大小的方法也有多种（即高度、宽度和高宽比：三者任选其二）。

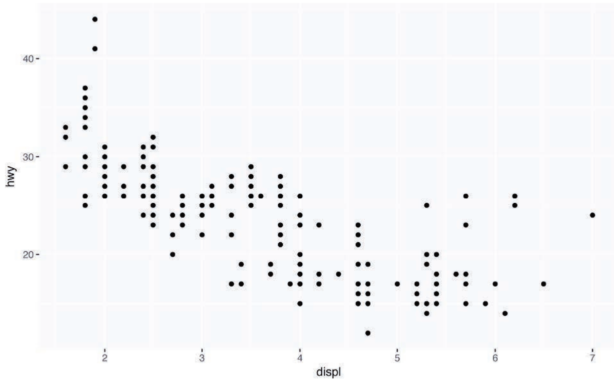
我们只使用以上 5 种选项中的 3 种。

- 我们发现，宽度一致的图形是最令人赏心悦目的。为了使图形宽度保持一致，我们设置图形的默认参数为 `fig.width = 6`（6 英寸）和 `fig.asp = 0.618`（黄金分割）。在单个代码段中，我们只调整 `fig.asp`。
- 我们使用 `out.width` 控制输出图形的大小，并将其设置为行宽的百分比。默认设置为 `out.width = "70%"` 和 `fig.align = "center"`。这样一来，图形既不会占用过多空间，也不会显得太拥挤。
- 如果想在一行中放置多个图形，可以将 `out.width` 设置为 `50%` 以放置 2 个图形、设置为 `33%` 以放置 3 个图形，或者设置为 `25%` 以放置 4 个图形，同时还要设置 `fig.align = "default"`。按照具体的说明方式（例如，是要展示数据还是要展示图形），我们也会根据情况调整 `fig.width`，后面还会继续讨论这点。

如果你发现必须眯起眼睛才能看清图形上的文本，那么就需要调整 `fig.width` 参数。如果 `fig.width` 大于最终文档中的图形的尺寸，那么文本就会显得过小；如果 `fig.width` 小于最终文档中的图形的尺寸，那么文本就会显得过大。一般来说，你需要试验几次才能找到 `fig.width` 和最终文档中的图形的最佳比例。为了说明这个问题，我们可以看看以下 3 幅图形，它们的 `fig.width` 分别是 4、6 和 8。







如果想要让所有图形中的字体都保持一致大小，那么只要设置了 `out.width`，就同时还需要调整 `fig.width`，使其与默认 `out.width` 保持同样的比例。例如，如果默认 `fig.width` 为 6，`out.width` 为 0.7，那么当设置 `out.width = "50%"` 时，你需要同时将 `fig.width` 设置为 4.3（6*0.5/0.7）。

21.7.2 其他重要选项

当代码和文本混合时，就像本书一样，我们建议你设置 `fig.show = "hold"`，以使图形显示在代码后面。这样做的好处是，可以强制使用解释性图形将大块代码分解成更小的部分。

如果想要为图形添加说明文字，可以使用 `fig.cap`。在 R Markdown 中，这样做会将图形从“内联”模式修改为“浮动”模式。

如果想要生成 PDF 格式的输出文件，使用默认的图形格式即可，因为默认格式就是 PDF。PDF 是一种良好的默认格式，因为它是一种高质量的向量化图形。但是，如果图形中包括几千个数据点，那么生成的图形就会很大，速度也会非常慢。这时可以设置 `dev = "png"` 来强制使用 PNG 图形格式。这种格式的图形质量稍差，但体积更小。

即使通常不为代码段添加标签，但为生成图形的代码段取名也是一种非常好的做法。代码段标签可以作为保存在磁盘上的图形文件的名称，因此如果为代码段取了名字，你就可以更容易地找出需要的图形，并在其他环境中继续使用（换言之，你可以非常迅速地将一个图形放入电子邮件或推文中）。

21.8 更多学习资源

学习更多相关知识的最好去处绝对是 ggplot2 教材：《ggplot2：数据分析与图形艺术》。这本书更加深入地介绍了基础理论，而且使用了更多的示例来介绍如何组合多个图形以解决实际问题。遗憾的是，这本书没有免费的在线版本，但你可以在 https://github.com/hadley/ggplot2-book 找到其源代码。

另一个绝好资源是 ggplot2 扩展指南网站（http://www.ggplot2-exts.org/）。这个站点列举了很多 ggplot2 扩展包，它们实现了新的几何对象和新的标度。如果想要做一些 ggplot2 难以实现的事情，不妨从这里开始。

第22章

R Markdown输出类型

22.1 简介

迄今为止，我们已经知道了 R Markdown 可以生成 HTML 文档。本章将简单介绍 R Markdown 的一些其他类型的输出结果。设置 R Markdown 文档的输出类型的方式有 2 种。

(1) 修改 YAML 文件头可以进行永久性设置。

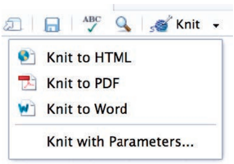
```
title: "Viridis Demo"
output: html_document
```

(2) 手动调用 `rmarkdown::render()` 函数可以进行临时性设置。

```
rmarkdown::render(
  "diamond-sizes.Rmd",
  output_format = "word_document"
)
```

如果想要编程实现多种类型的输出，那么就需要使用这种设置方式。

RStudio 中的 knit 按钮可以按照 `output` 域中列出的第一种格式来输出文件。点击 knit 按钮旁的下拉菜单，也可以生成其他相应格式的输出。



22.2 输出选项

每种输出类型对应一个 R 函数，使用 `foo` 或 `pkg::foo` 的函数调用形式都可以。如果省略 `pkg`，R 会默认使用 rmarkdown 包。知道输出函数的名称是非常重要的，因为可以通过函数名获取帮助。例如，如果想要知道可以在 `html_document()` 函数中使用哪些参数，可以使用 `?rmarkdown::html_document()` 来查看帮助。

如果不想使用默认参数值，可以使用扩展的 `output` 域。例如，如果想要生成一个带有浮动表格内容的 `html_document` 文档，可以设置如下：

```
output:
  html_document:
    toc: true
    toc_float: true
```

甚至还可以提供一个格式列表，以生成多种输出：

```
output:
  html_document:
    toc: true
    toc_float: true
  pdf_document: default
```

如果不想覆盖任何默认选项，注意一下特殊的语法。

22.3 文档

上一章重点介绍了默认函数 `html_document()` 的输出，这个函数还有几个基本的变体，可以生成不同类型的文档。

- `pdf_document` 可以使用 LaTeX（一种开源的文档排版系统）生成 PDF 文档。LaTeX 需要安装，如果没有安装，RStudio 会提醒你。
- `word_document` 可以生成 Microsoft Word 文档（.docx）。
- `odt_document` 可以生成 OpenDocument Text 文档（.odt）。
- `rtf_document` 可以生成 Rich Text Format 文档（.rtf）。
- `md_document` 可以生成 Markdown 文档。这种文档本身不是很有用，但你可以将其与其他系统结合起来使用，如 CMS 系统或实验室 wiki。
- `github_document` 是 `md_document` 的一种定制版本，前者生成的文档可以在 GitHub 上分享。

记住，当为决策者生成文档时，你应该在 `setup` 代码段中修改全局设置，关闭默认的显示代码选项：

```
knitr::opts_chunk$set(echo = FALSE)
```

对于 `html_document` 文档，另一个选项可以让代码段默认是隐藏的，但点击鼠标后可见：

```
output:
  html_document:
    code_folding: hide
```

22.4 笔记本

笔记本文件是由 `html_document` 的变体 `html_notebook()` 函数生成的。虽然这两种函数生成的文件在格式上非常相似，但目的不同。`html_document` 主要用于与决策者进行沟通，而笔记本的主要用途则是与其他数据科学家协同工作。由于目的不同，HTML 输出文件的使用方式也不相同。这两种 HTML 输出文件都会包含全部结果，但笔记本中还包括完整的源代码。这意味着我们可以通过两种方式使用由 `html_notebook()` 生成的 .nb.html 文件。

- 你可以使用网页浏览器来查看这种文件和分析结果。与 `html_document` 不同，这种文件始终包含了生成文件的源代码的一个嵌入式副本。
- 你还可以在 RStudio 中编辑这种文件。当打开一个 .nb.html 文件时，Rstudio 会自动重建生成它的 .Rmd 文件。将来你还可以在这种文件中包含支持文件（如 .csv 数据文件），需要时可以自动提取出这些支持文件。

通过电子邮件发送 .nb.html 文件是与同行分享分析结果的一种简单方式。但是，如果别人想修改这个文件，事情就变得不妙了。如果出现了修改文件的需求，是时候学习 Git 和 GitHub 了。学习 Git 和 GitHub 的初期会非常痛苦，但合作带来的回报是非常丰厚的。正如我们在前面提到的，Git 和 GitHub 已经超出了本书的讨论范围，但如果你已经开始使用它们，我们会给你一个非常有用的提示：需要同时使用 `html_notebook` 和 `github_document` 两种输出形式。

```
output:
  html_notebook: default
  github_document: default
```

`html_notebook` 可以在本地预览，还可以通过电子邮件进行分享。`github_document` 可以创建一个最简单的 MD 文件，你可以将其提交到 Git 中。你可以清晰地看到分析结果（不只是代码）是如何随时间的推移而不断发展的，GitHub 会在网上很好地呈现出这个过程。

22.5 演示文稿

R Markdown 也可以生成演示文稿。与 Keynote 或 PowerPoint 这样的工具相比，R Markdown 没有那么多视觉效果和控制方式，但它可以自动将 R 代码的结果插入演示文稿，这样可以节省大量时间。制作演示文稿的方法是将内容分为多个幻灯片，在每个一级标题（`#`）或二级标题（`##`）处开始一个新幻灯片。你也可以不使用标题来划分幻灯片，而是通过插入一条水平分隔线（`***`）来创建一张新的幻灯片。

R Markdown 可以生成 3 种内置格式的演示文稿。

`ioslides_presentation`
: ioslides 格式的 HTML 演示文稿。

`slidy_presentation`
: W3C Slidy 格式的 HTML 演示文稿。

beamer_presentation

LaTeX Beamer 格式的 PDF 演示文稿。

此外，由其他 R 包提供的另外 2 种常用的演示文稿格式如下所示。

revealjs::revealjs_presentation

reveal.js 格式的 HTML 演示文稿。需要安装 revealjs 包。

rmdshower

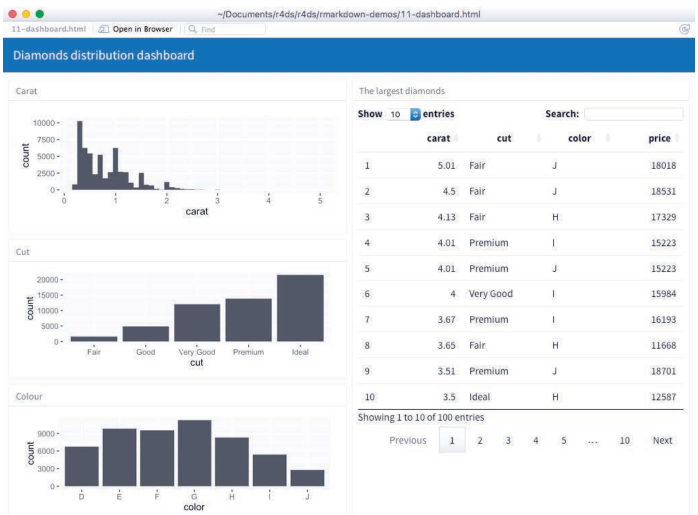
提供了 shower 演示引擎的一个包装器。

22.6 仪表盘

仪表盘是沟通大量数据的一种快速而又直观的有效方法。flexdashboard 包可以非常轻松地使用 R Markdown 文件来创建仪表盘，它确定了使用标题控制仪表盘布局的约定。

- 每个一级标题（#）都可以在仪表盘中创建一个新页。
- 每个二级标题（##）都可以创建一个新列。
- 每个三级标题（###）都可以创建一个新行。

例如，要想创建以下这个仪表盘：



可以使用以下代码：

```
---
title: "Diamonds distribution dashboard"
output: flexdashboard::flex_dashboard
---

```{r setup, include = FALSE}
library(ggplot2)
```
```

```
library(dplyr)
knitr::opts_chunk$set(fig.width = 5, fig.asp = 1/3)
```

Column 1

Carat

```{r}
ggplot(diamonds, aes(carat)) + geom_histogram(binwidth = 0.1)
```

Cut

```{r}
ggplot(diamonds, aes(cut)) + geom_bar()
```

Color

```{r}
ggplot(diamonds, aes(color)) + geom_bar()
```

Column 2

The largest diamonds

```{r}
diamonds %>%
  arrange(desc(carat)) %>%
  head(100) %>%
  select(carat, cut, color, price) %>%
  DT::datatable()
```
```

flexdashboard 还提供了一些简单工具，用于创建工具栏、标签页、输入框和标尺。要想学习更多关于 flexdashboard 的知识，可以访问 http://rmarkdown.rstudio.com/flexdashboard/。

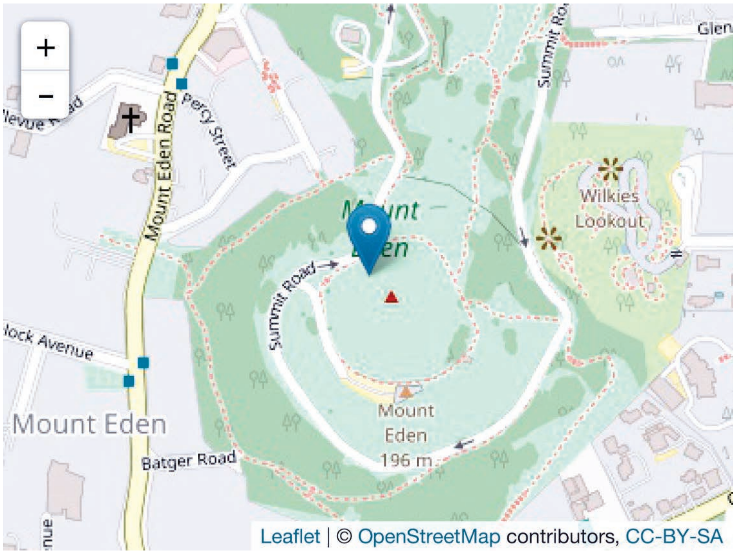
# 22.7 交互元素

HTML 格式（文档、笔记本、演示文稿或仪表盘）的所有文件都可以包含可交互组件。

## 22.7.1 htmlwidgets

HTML 是一种可交互的格式，要想利用这种交互性，你可以使用 htmlwidges，这是能够生成 HTML 可视化元素的一组 R 函数。例如，它可以生成以下的 leaflet 地图。如果在浏览器上查看这一页，就可以对地图进行拖动、放大、缩小等操作。显然，不能对图书进行这种操作，因此 rmarkdown 自动插入了一张静态的屏幕截图：
```

```
library(leaflet)
leaflet() %>%
  setView(174.764, -36.877, zoom = 16) %>%
  addTiles() %>%
  addMarkers(174.764, -36.877, popup = "Maungawhau")
```



htmlwidgets 的优势是，无须了解任何关于 HTML 和 JavaScript 的知识就可以使用它们。因为所有细节都封装在包中，所以根本不需要关心细节。

以下各包都可以提供 htmlwidgets。

- dygraphs 可以实现交互式的时间序列可视化。
- DT 可以实现交互式表格。
- rthreejs 可以实现交互式三维图表。
- DiagrammeR 可以实现示意图（如流程图和简单的节点关联图）。

要想学习更多关于 htmlwidgets 的知识或查看提供 htmlwidgets 的完整 R 包列表，可以访问 http://www.htmlwidgets.org/。

22.7.2 Shiny

htmlwidgets 提供了**客户端**的交互性——所有交互都发生在浏览器中，与 R 无关。这种方式有很明显的优点，因为你可以自由地分发 HTML 文件，不用考虑与 R 的连接。但这种方式的基本局限性在于，所有功能都要依靠 HTML 和 JavaScript 来实现。实现交互性的另一种方式是使用 Shiny，这个包可以使用 R 代码创建交互元素，而不用依靠 JavaScript。

如果想在 R Markdown 文档中调用 Shiny 代码，需要在文件头中添加 `runtime::shiny`：

```
title: "Shiny Web App"
output: html_document
runtime: shiny
```

然后我们可以使用“输入”函数向文档中添加交互元素：

```
library(shiny)

textInput("name", "What is your name?")
numericInput("age", "How old are you?", NA, min = 0, max = 150)
```

接下来，我们可以使用 `input$name` 和 `input$age` 来引用交互元素的值，如果值发生了改变，使用这些值的代码会自动重新运行。



我们无法在这里展示一个实际的 Shiny 应用，因为 Shiny 的交互发生在**服务器端**。这意味着你可以在不使用 JavaScript 的情况下编写交互式应用，但需要一个服务器来运行这个应用。这就导致了一个逻辑问题：Shiny 应用需要一个 Shiny 服务器才能在线运行。当你在自己的计算机上运行 Shiny 应用时，Shiny 会自动建立一个 Shiny 服务器，但如果你想将这种交互式应用在网上公之于众，就需要面向公众的一个 Shiny 服务器。这就是使用 Shiny 时需要考虑的一个基本问题：可以在 Shiny 文件中实现 R 的所有功能，但代价是需要运行 R 的一个服务器。

要想学习更多关于 Shiny 的知识，可以访问 http://shiny.rstudio.com/。

22.8 网站

如果再有一点基础设施，我们还可以使用 R Markdown 生成一个完整的网站。

- 将你的 .Rmd 文件放进一个单独的目录。index.Rmd 就可以作为网站的主页。
- 添加一个名为 _site.yml 的 YAML 文件来提供站点导航。例如：

```
name: "my-website"
navbar:
  title: "My Website"
  left:
    - text: "Home"
      href: index.html
    - text: "Viridis Colors"
      href: 1-example.html
    - text: "Terrain Colors"
      href: 3-inline.html
```

运行 `rmarkdown::render_site()` 函数来建立 _site 目录，这个目录中包含了一个独立静态网站所需的所有文件。你还可以使用 RStudio Project 来建立网站目录，RStudio 会在 IDE 中添加一个 Build 标签页，你可以使用这个标签页来建立和预览自己的站点。

要想学习更多这方面的知识，可以访问 https://rmarkdown.rstudio.com/rmarkdown_websites.html。

22.9 其他类型

其他包可以生成更多输出类型。

- bookdown 包可以使得编写图书更加容易，比如编写像本书这样的一本图书。如果想要了解更多，可以阅读谢益辉的著作 *Authoring Books with R Markdown*。当然，这本书也是通过 bookdown 编写的。R 社区中有很多利用 bookdown 编写的图书。
- prettydoc 包提供了具有多个漂亮主题的轻量级文档格式。
- rticles 包可以将 R Markdown 文件编译成一些科学杂志要求的特殊格式。

要想获得更多格式的列表，可以访问 http://rmarkdown.rstudio.com/formats.html。你也可以通过 http://bit.ly/CreatingNewFormats 上的操作指南创建自己的格式。

22.10 更多学习资源

如果想要获取更多知识，以使用不同类型的输出进行有效沟通，那么我们推荐以下学习资源。

- 如果想要提高演示和表达能力，我们推荐你阅读 Neal Ford、Matthew McCollough 和 Nathaniel Schutta 所著的 *Presentation Patterns*。这本书提供了（从低级到高级的）一整套有效的模式，可以帮助你提高自己的演示和表达能力。
- 如果想要进行学术讨论，我们推荐你阅读一下 *Leek group guide to giving talks* 这本书。
- 虽然我们没有亲身体验过，但 Matt McGarrity 关于公开演讲的在线课程（https://www.coursera.org/learn/public-speaking）确实具有良好的口碑。
- 如果想要创建大量仪表盘，可以阅读 Stephen Few 所著的 *Information Dashboard Design: The Effective Visual Communication of Data*。这本书会帮助你创建真正有价值的仪表盘，而不只是虚有其表的仪表盘。
- 平面设计知识经常有助于人们想法的有效沟通。《写给大家看的设计书》[1] 是非常好的一本入门级图书。

注 1：全球销量百万册的设计入门书，简洁实用，详细介绍可见图灵社区本书第 4 版的介绍页面：ituring.com.cn/book/1551。——编者注

第23章

R Markdown工作流

我们在前面讨论过一个基本的工作流，首先说明了如何在**控制台**中交互地使用 R 代码，然后介绍了**脚本编辑器**的功能。R Markdown 将控制台和脚本编辑器结合在一起，既可以进行交互式数据探索，也可以长久保存代码。你可以在一个代码段内快速迭代，先修改代码，然后使用 Ctrl+Shift+Enter 重新运行代码。如果愿意的话，你还可以继续添加新的代码段。

R Markdown 的重要性还在于，它可以将文本和代码紧密地集成在一起。这使得它既可以开发代码，又可以记录你的想法，是一种非常棒的**分析式笔记本**。自然科学研究中一般都会有个实验记录本，分析式笔记本的一些用途与实验记录本是基本相同的。

- 记录你做了什么，以及为什么要这样做。不管记忆力多么强大，如果不进行记录，你总会有忘记重要事情的时候。好记性不如烂笔头！
- 帮助你进行更加缜密的思考。如果能在工作过程中随时记录想法并不断进行反思，那么就更有可能将分析工作做得更好。在最后归纳分析结果以便与他人分享时，这些记录还可以节省你大量的时间。
- 帮助人们理解你的工作。数据分析通常不是一个人的事情，你经常会作为团队中的一员。实验记录本不但可以告诉同事或同实验室的人你做了哪些工作，还可以告诉他们你为什么要这样做。

有效使用实验记录本的一些成熟经验完全可以推广到分析式笔记本上。根据自己的亲身体会，并结合 Colin Purrington 关于实验记录本的意见，我们给出了如下一些建议。

- 确保每个笔记本都有一个描述性标题和一个有助于记忆的文件名，并在第 1 段中简要地介绍一下这项分析的主要目的。
- 使用 YAML 文件头中的日期域来记录开始使用这个笔记本的日期：

```
date: 2016-08-23
```

使用 ISO 8601 标准的 YYYY—MM—DD 格式来避免歧义。即使通常不使用这种格式的日期，你也一定要这样做！

- 对一个分析思路花费大量时间后，如果发现还是走入了死胡同，此时不要丢弃它。进行简短的笔记，记录为什么会失败，并保存在笔记本中。如果未来的某个时刻回过头来再进行这项分析时，你就可以避免重蹈覆辙。
- 一般来说，我们不使用 R 进行数据录入。但如果你需要记录一小段数据的话，可以使用 `tibble::tribble()` 函数来录入，它非常简单直观。
- 如果在某个数据文件中发现了一个错误，千万不要直接修改，而是应该通过编写代码来修改错误值，并解释为什么要进行这个修改。
- 在结束一天的工作前，请确认你的笔记本可以正确生成（如果使用了缓存，一定要清除）。这样可以让你趁热打铁地修改笔记本中可能存在的错误。
- 如果想让代码长期可重用（也就是说，一个月或一年后这段代码依然可以运行），你需要跟踪代码中使用的 R 包的版本更新信息。一种非常好的方法是使用 packrat 包，它可以将 R 包保存到你的项目目录中。另一种方法是使用 checkpoint 包，它可以在一个特定日期重新安装所有可用的 R 包。还有一种很简便的方法，但是既不够正规，也不够优雅，就是在笔记本中加入运行 `sessionInfo()` 函数的一个代码段，虽然这样不能很方便地更新 R 包，但至少可以让你知道这些 R 包现在的版本。
- 你的整个职业生涯中会创建很多很多分析式笔记本。应该如何管理它们，才能在以后找到需要的笔记本呢？我们的建议是，将它们保存在一个独立的项目中，并使用一种恰当的命名模式。

作者简介

Hadley Wickham 是 RStudio 首席科学家、R Foundation 资深成员。他创建的很多工具（包括计算工具和认知工具）使得数据科学变得更容易、更快速，同时也更有趣。他的作品包括用于数据科学的多个软件包（tidyverse、ggplot2、dplyr、purrr、readr 等），以及一些工具软件（roxygen2、testthat、devtools）。此外，他还是作家、教育者，同时也是不懈推动使用 R 进行数据科学的活跃演讲者。

Garrett Grolemund 是任职于 RStudio 的统计学家、教师和 R 开发者。他开发了著名的 R 包 lubridate，并著有《R 语言入门与实践》一书。

Garrett 是 DataCamp.com 和 oreilly.com/safari 中广受欢迎的一名 R 语言导师，曾受邀在很多公司讲授 R 语言和数据科学，这些公司包括 Google、eBay、Roche 等。Garrett 在 RStudio 组织了各种在线研讨会和专题讨论会，并编写了多个备受称赞的 R 语言速查表。

封面简介

本书封面上的动物是一只鸮鹦鹉（学名：Strigops habroptilus），它是原产于新西兰的一种不会飞的大型鸟类。成年鸮鹦鹉可以长到 64 厘米高，体重可达 4 千克。它们的羽毛通常为黄绿色，但个体间的差异非常大。鸮鹦鹉是夜行动物，依靠灵敏的嗅觉在夜间活动。虽然没有飞行能力，但鸮鹦鹉具有强壮的双腿，这使得它们的奔跑能力和攀爬能力远远强于多数鸟类。

鸮鹦鹉又称为 kakapo，这个名称来自于新西兰原住民毛利人的语言。鸮鹦鹉是毛利文化的重要组成部分，既是毛利人的食物来源，也是毛利神话的一部分。鸮鹦鹉的皮和羽毛还可以用来制作斗篷和披肩。

因为一些食肉动物在欧洲殖民运动期间被带到了新西兰，所以鸮鹦鹉现在濒临灭绝，现存数量已不足 200 只。新西兰政府已经在没有食肉动物的 3 个岛屿上建立了特别保护区，以此来积极地恢复鸮鹦鹉的数量。

O’Reilly 图书封面上的很多动物都是濒危物种，它们对整个世界都很重要。如果你想为保护动物做些贡献，请访问 animals.oreilly.com。

封面照片来自 *Wood’s Animate Creations*。

O'REILLY®

R数据科学

R社区领军人物作品，从典型数据科学项目所需工具模型着手，带领读者掌握R语言精华，学会熟练使用多种工具解决各种数据科学难题。

- **探索——以可视化作为R编程起点，再进行重要变量选取、筛选关键观测等重要数据操作，并对数据提出问题且找到答案。**
- **处理——导入、整理并转换数据。**
- **编程——管道操作的工作原理和替代方式，函数使用规则，如何实现迭代。**
- **模型——深刻理解模型背后的数学理论和数据，直观认识统计模型工作原理。**
- **沟通——学会R Markdown，让人们快速轻松理解你的工作。**

哈德利·威克姆（Hadley Wickham），RStudio首席科学家，统计学家，斯坦福大学、奥克兰大学、莱斯大学兼职统计学教授。已被下载数百万次的ggplot2等多款知名R包的开发者，一直致力于让普罗大众更容易上手数据分析，被R社区誉为“改变了R的人”。另著有《R包开发》等书。

加勒特·格罗勒芒德（Garrett Grolemund），RStudio数据科学家，知名R培训师，曾受邀在Google、eBay等诸多公司讲授R语言和数据科学，在DataCamp开授的R相关课程备受R开发者喜爱。另著有《R语言入门与实践》。

“Hadley Wickham是数据科学领域内的传奇人物，发明了一套全新的数据分析方法。他与Garrett Grolemund合著的这本书详细阐明了这种新方法，被数据分析者奉为圣经。”

——**Roger D. Peng**

约翰霍普金斯大学教授

“建议初次接触R的人都看看这本书中的R代码。作者详细说明了在R中处理数据的基本原则。”

——**J.J.Allaire**

RStudio创始人、CEO

DATA ANALYSIS/STATISTICAL SOFTWARE

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机 / 数据科学 / R

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-48639-4
定价：139.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks